THE UNIVERSITY OF QUEENSLAND

Development of a 2D Unstructured Grid Generator for Eilmer4 Using an Advancing Front Algorithm

Student Name: Matthew TRUDGIAN

Course Code: ENGG7240

Supervisor: Dr. Peter Jacobs

Submission date: 27 October 2017

Faculty of Engineering, Architecture and Information Technology

Abstract

Unstructured grids have been becoming increasingly relevant to Computation Fluid Dynamics solvers over recent years. The University of Queensland's in-house solver, Eilmer4, has recently been updated to simulate flows over unstructured grids. Eilmer4 currently only has a limited toolset of functions for generating unstructured grids and these tools will be discussed along with current state of the art gridding techniques. An Advancing Front Algorithm is developed that produces successful gridding results for several test cases. The algorithm's shortcomings are discussed and potential for future development of the algorithm are outlined. The algorithm provides an important first step to developing a robust unstructured grid generator for Eilmer4.

Contents

List of Figures

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) is a field of research focused on simulating the flow of fluids around objects using numerical methods. This approach uses a series of boundaries to describe the flow-field and declares boundaries to be for example, a solid wall, an inflow, or an outflow, among many others. A mesh is then created that divides up this bounded flow-field into small, finite-volume cells, which are then solved iteratively using the Navier-Stokes equations. Fortunately, this report will focus exclusively on the generation of the grid rather than solving these equations.

Originally structured grids were used for simulating fluid dynamics, however more recently unstructured grids have started to become dominant. Due to the continued increase in computational power, CFD simulations are becoming more ambitious in the geometries that they wish to solve. This means that the grids being generated for current CFD applications are becoming exceptionally complex, including more surface features and having significantly larger flow-fields. The computational overhead of implementing an unstructured grid due to its requirement for connectivity tables, has been out scaled by the increasing performance of the computing components. In light of the increasing power of computation, the UQ in-house CFD solver Eilmer4 has recently been updated to be able to handle unstructured grids. In its current state however, it has very limited tools available to construct unstructured grids on its own. The Eilmer4 Geometry User Guide(2017)[2] claims that there are only three methods available for constructing an unstructured grid, which are:

- Re-describe a Structured Grid as an Unstructured Grid
- Import a grid in SU2 format from a third-party program, namely Pointwise
- Use Heather Muir's[10] paver, which is a work in progress

The scope of this thesis therefore will be to develop an initial 2D unstructured grid algorithm that can be implemented into the Eilmer4 unstructured grid constructer. This will reduce

Eilmer4's reliance on third-party grid generation software, and increase the capabilities of the solver. The thesis will investigate the current options that are available in the unstructured gridding literature and implement one of these strategies. The aim is to be able to construct preliminary grids that can be refined in future development upon this algorithm.

Chapter 2

Literature Review

Before the unstructured grid generator was developed, a thorough literature review was conducted. This review will begin by looking at when unstructured grids are preferred over structured grids in different CFD applications and their benefits. It will then consider the different methods for generating unstructured grids in two dimensions and make mention of the methods that were found to be unsuitable for this application. Once these methods have been described the review will look into the Advancing Front Technique in more detail. Finally, it will discuss the potential implementations of the refinement stage that comes after the initial advancing front grid has been constructed. The refinement stage is outside the scope of the thesis however it is important to understand why the grids that will be generated using this algorithm are so coarse.

2.1 Uses of Unstructured Grids in CFD

For the majority of CFD's history structured grids have been the dominant form of grid type for simulations. They perform better computationally because they save on time and memory by having their cells ordered implicitly. This negates the need for connectivity tables and allows the cells to be stored in arrays in memory. Pointwise(2017) claims that the resolutions of structured grids are generally superior, and the cell alignment is better and produces more accurate results in most cases. Structured grids do however have downfalls. Most significantly in cases where boundary layer clustering is required. These clusters of cells need to propagate all the way to the boundary of the mesh which produces a considerable number of cells out in the flow-field. This in turn consumes a substantial amount of computational power for cells that have no important flow dynamics contained within them. A good example of this cell clustering issue is shown by the gridding strategy used by Pointwise(2017) in Fig. 1, which shows the large number of clustered cells in the outer flow-field.

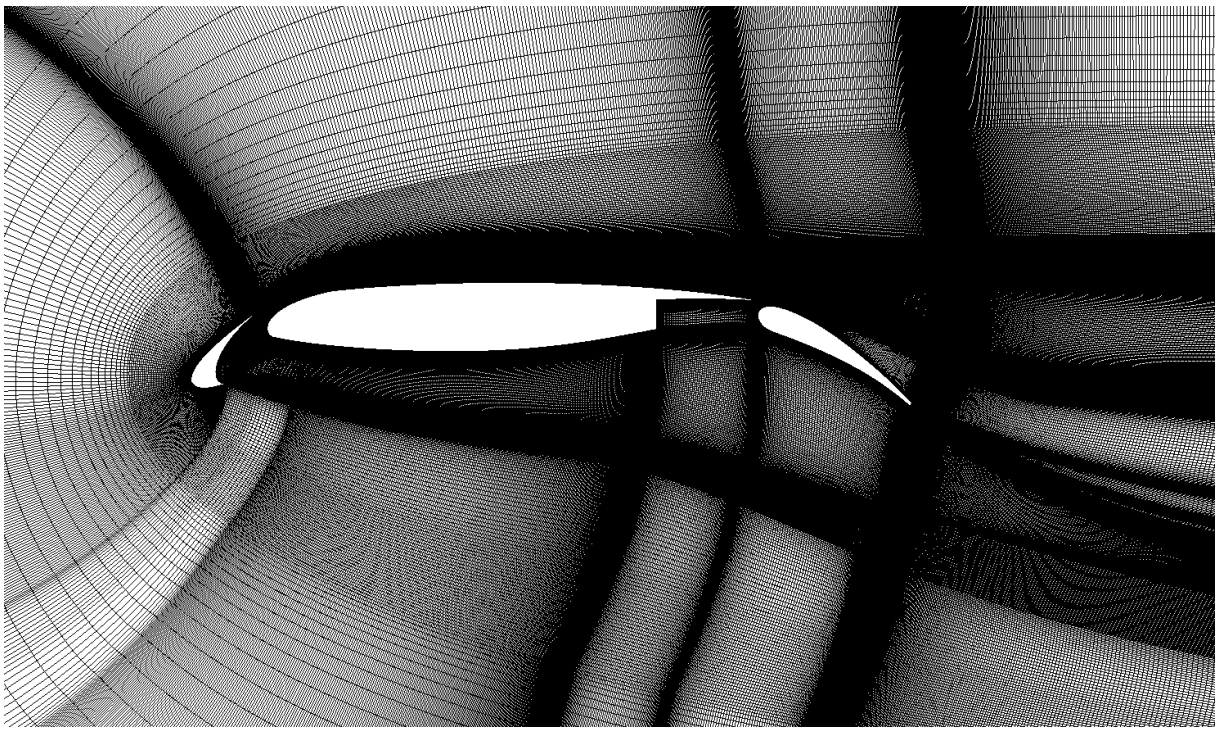


Figure 1 - Figure taken from Pointwise[4] showing a Chimera mesh over a complicated aerofoil. Note the significant number of cells in the outer flow-field that are a result of requiring a high-resolution boundary layer mesh.

Unstructured grids aim to reduce the number of cells in the outer flow-field while still maintaining the required resolution to numerically solve the flow at the boundary layer level. The increase in computational resources required to handle unstructured grids connectivity tables is surpassed by the efficiencies it provides by having fewer cells. Figure 2 shows the same aerofoil geometry but this time using an unstructured grid.

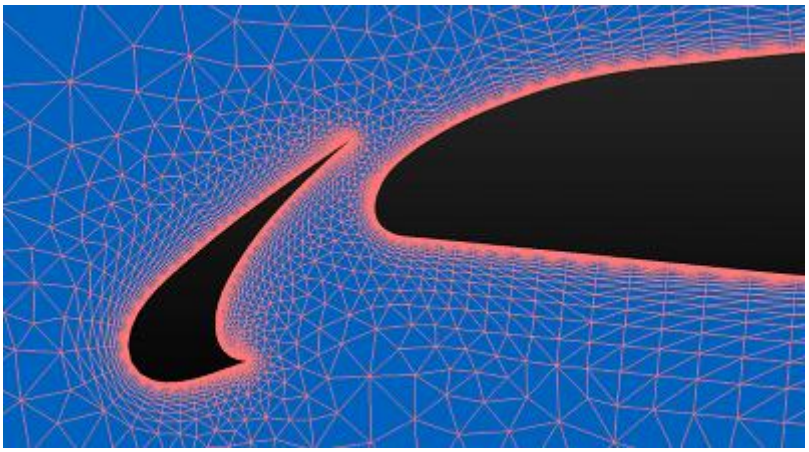


Figure 2 - Figure taken from Pointwise[5] showing the benefit of creating an unstructured grid for the flow-field while keeping the boundary layer grid structured.

The number of cells falls off dramatically as they move further away from the clustered boundary layer. Keeping this in mind, the automated algorithm should be able to produce grids that can propagate out to form a flow-field.

This is not the only situation in which unstructured grids are powerful. As Pointwise(2017) points out unstructured grids are very powerful when dealing with complex geometries, as they are able to alter their cell width and size more freely. This is due to the superior tessellation ability of triangular cells. Using unstructured grids in these cases produce grids with higher quality cells than their structured counterparts and therefore allow for more accurate simulations.

2.2 Methods for Generating Unstructured Grids

There are several methods that can be used to generate unstructured grids. The main three are the Octree Method, Delaunay Method, and Advancing Front Method. Both the Octree and Delaunay methods were found to be unsuitable for this implementation because they are both more appropriate to three-dimensional gridders, as they are much more resilient to cell overlap. For the two-dimensional case however, Löhner(2008) suggests the advancing front method to be the most robust for creating the initial unstructured grid.

2.2.1 Octree and Quadtree Methods

Octree and Quadtree are methods used for generating grids in both three and two dimensions respectively. Due to the thesis scope, only the two-dimensional Quadtree method will be investigated. It involves subdividing squares contained by the boundaries until the entire bounded section has been gridded. Each successive square is produced at half the size of the previous square to preserve cell quality during the cell size transition. Figure 3 taken from Owen(n.d.)[6] shows an implementation of the Quadtree method.

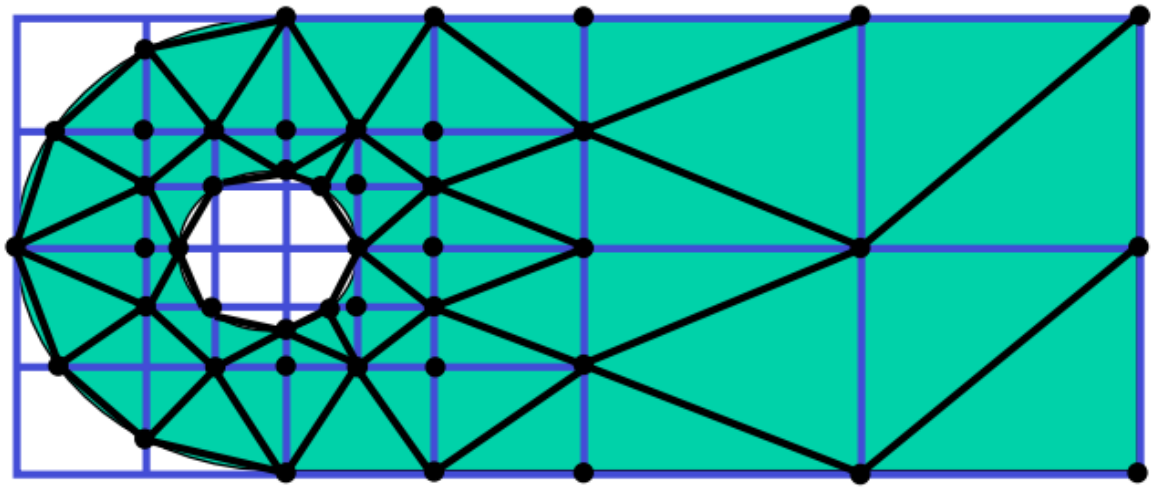


Figure 3 - Quadtree implementation as shown by Owen(n.d.). The octree squares provide the initial points for the tri cells to connect to.

The vertices of these squares act as nodes with which to connect tri cells. It produces grids with a guarantee of no cell overlap, however the clustering of cells around areas of complex geometry isn't a desirable trait of the algorithm.

2.2.2 Delaunay Method

The Delaunay method is the second type of grid generation algorithms that exist in literature. Löhner(2008) describes an algorithm that creates a grid that iteratively introduces new points that have been pre-described by a structured background grid. Each time a new point is introduced, all elements that contain the point in their circumcircle are deleted, and the cavity is repopulated by new Delaunay cells. Figure 4 shows a visual representation put forward by Löhner which describes how a grid is generated using this method.

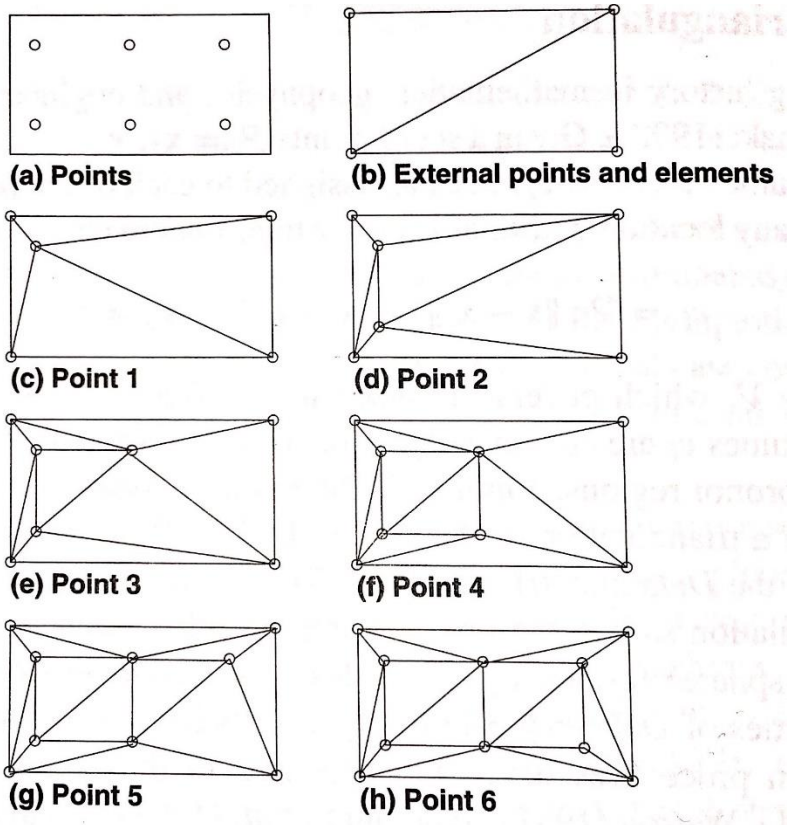


Figure 4 - Figure taken from Löhner(2008) which describes how a Delaunay Algorithm functions. First requiring a set of pre-existing points and then iteratively introducing them and re-describing the mesh each time.

The downside to this method is that it requires a structured background grid to generate an unstructured grid from scratch. This was ultimately decided to be an insufficient method to use for the entirety of the grid generation algorithm. However, Mavriplis(1997)[7] does present an opportunity to use it as a post-processing tool for grid refinement by using cells that have already been generated from an unstructured grid instead of the background structured grid.

2.3 Advancing Front Method

The Advancing Front method is the final type of grid generation algorithm. This method works by describing the boundary of the grid-space and dividing the boundaries up into discrete faces. These faces are then activated to form an active front, with the intention of them propagating into the interior grid-space. For each iteration, an active face is selected based upon its size with the smallest being preferred (Löhner, 2008). This active face then defines a point in the interior grid-space and creates a new cell. The active face is then deleted and the new faces that it generates become active. Mavriplis presents an intuitive illustration of how an advancing front propagates into a bounded domain. Figure 5 shows a partially advanced front with the option to create either cell A or cell B depending on whether it is desired to use a new or existing point.

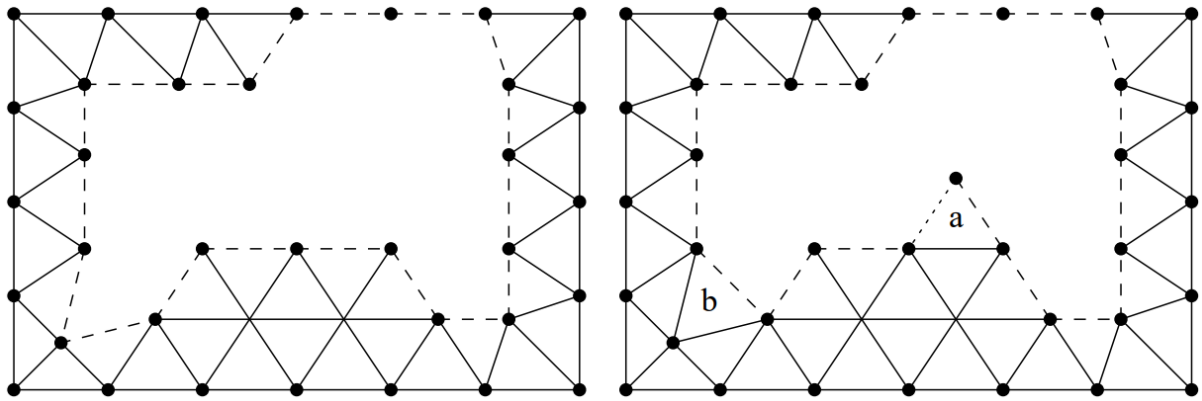


Figure 5 - Figure taken from Mavriplis(1997) which shows a partially solved Advancing Front Algorithm. The image to the right shows two options for cell advancement. Cell a represents a new cell using a new point, where Cell b will be using an existing point.

The Advancing Front Algorithm then proceeds to generate cells until the entire front has been completed. Löhner(2008) discusses a pseudo-code algorithm demonstrating how to implement such a method in code. The following has been taken directly from Löhner.

1. Define the boundaries of the domain to be gridded.
2. Define the spatial variation of element size, stretchings and stretching directions for the elements to be created. In most cases, this is accomplished with a combination of background grids and sources as outlined above
3. Using the information given for the distribution of element size and shape in space and the line definitions, generate sides along the lines that connect surface patches. These sides form an initial front for the triangulation of the surface patches.

4. Using the information given for the distribution of element size and shape in space, the sides already generated and the surface definition, triangulate the surfaces. This yields the initial front of faces.
5. Find the generation parameters (element size, element stretchings and stretching directions) for these faces.
6. Select the next face to be deleted from the front; in order to avoid large elements crossing over regions of small elements, the face forming the smallest new element is selected as the next face to be deleted from the list of faces.
7. For the face to be deleted:
    a. Select a 'best point' position for the introduction of a new point *ipnew*;
    b. Determine whether a point exists in the already generated grid that should be used in lieu of the new point; if there is such a point, set this point to *ipnew* and continue searching;
    c. Determine whether the element formed with the selected point *ipnew* crosses any given faces; if it does, select a new point as *ipnew* and try again
8. Add the new element, point and faces to their respective lists
9. Find the generation parameters for the new faces from the background grid and the sources
10. Delete the known faces from the list of faces.
11. If there are any faces left in the front go to 6.

Löhner places significant emphasis on the prioritization of the smallest face being the most important to propagate first. This feature was implemented into this thesis algorithm's design.

One of the greatest problems of the Advancing Front Method is its tendency to create overlapping cells without proper oversight. Foucault et. al. describes a method for dealing with active faces that have the potential to create overlapping cells. This method describes implementing six special cases that use a pre-described angle condition to propagate cells using existing points. It was proven to be successfully implemented in their case of gridding complex surface geometries and so a similar implementation will be used for this project. Figure 6 shows the complete list of the six cases and the cells that are generated for each.

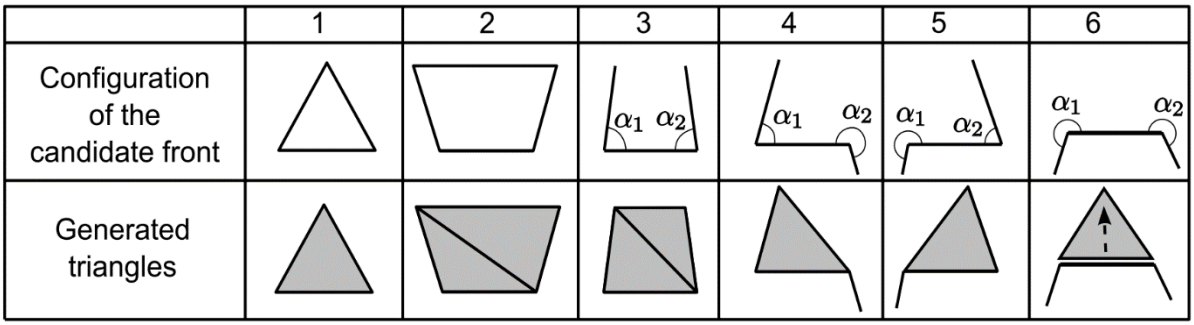


Figure 6 - Figure taken from Foucault et. al. which shows six cases whereby cells are created through pre-described angle cases.

The angles defined here to be α_1 and α_2 are intended to be chosen arbitrarily by the designer through an iterative method to find the best gridding results.

2.4 Grid Refinement and Post-Processing

As important as the initial grid generation is, there is still a second stage to the unstructured grid generation which refines the grid to achieve an optimal cell quality for the mesh. This thesis is only concerned about the initial stage of grid generation; however, several refinement strategies have been investigated to assist with places to start for future development of the algorithm.

2.4.1 Delaunay Triangulation Refinement

The Delaunay Refinement method uses a similar approach to the Delaunay Grid Generation method. However, instead of using an array of preexisting points, it uses the cells that have been generated by some initial gridding algorithm. In general, it's useful for refining cells that have been generated to be too large. In the case of Mavriplis(1997)[7] this technique was used to refine a grid that was generated via an advancing front. Figure 7 shows the implementation of this refinement strategy.

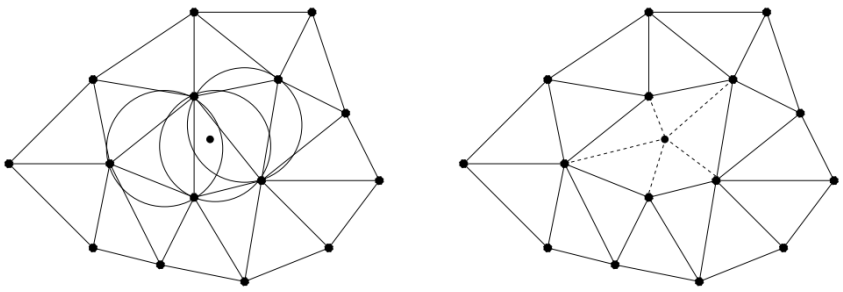


Figure 7 - Figure taken from Mavriplis(1997) showing how the Delaunay Refinement method selects the new point for the grid.

The method uses the Bowyer-Watson(1981)[8-9] algorithm for determining where the new point should be placed to ensure that the Delaunay condition is met. For this it uses the circumcircles of the three cells that it is trying to refine, and finds a point that is contained within all three. This point should be in the centermost point of the three circumcircles to assist with the cell quality of the refined cells. Mavriplis(1997)[7] then goes on to describe how to ensure that all large cells have been refined using a simple technique. This technique defines a maximum circumcircle for a cell and defines all cells that have circumcircles too large for this to be put in a list. The list is then iteratively solved making use of the Bowyer-Watson algorithm, until all cells have been refined.

2.4.2 Cell Removal

In cases where there are two poorly gridded cells that border each other, there is a technique that can reduce their impact on the mesh. This technique is outlined by Löhner(2008) which involves simply removing the poor quality cells and rejoining the surrounding ones. This is demonstrated in Fig. 8 and is extremely efficient.

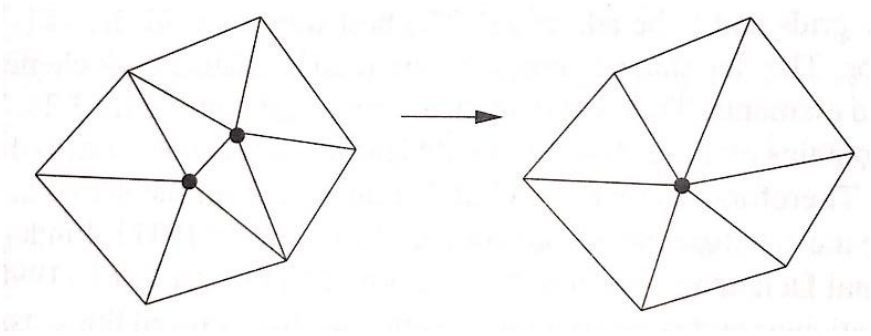


Figure 8 - Figure taken from Löhner(2008) showing how a Cell Removal is performed.

2.4.3 Face-Swapping

The final and most powerful refinement strategy is the Face-Swapping or Diagonal-Swapping technique outlined by Löhner(2008). This strategy is very simple, whereby you swap the faces between a given number of cells such that they change vertices. Figure 9 shows the number of different possible configurations for a group of cells to use Face-Swapping.

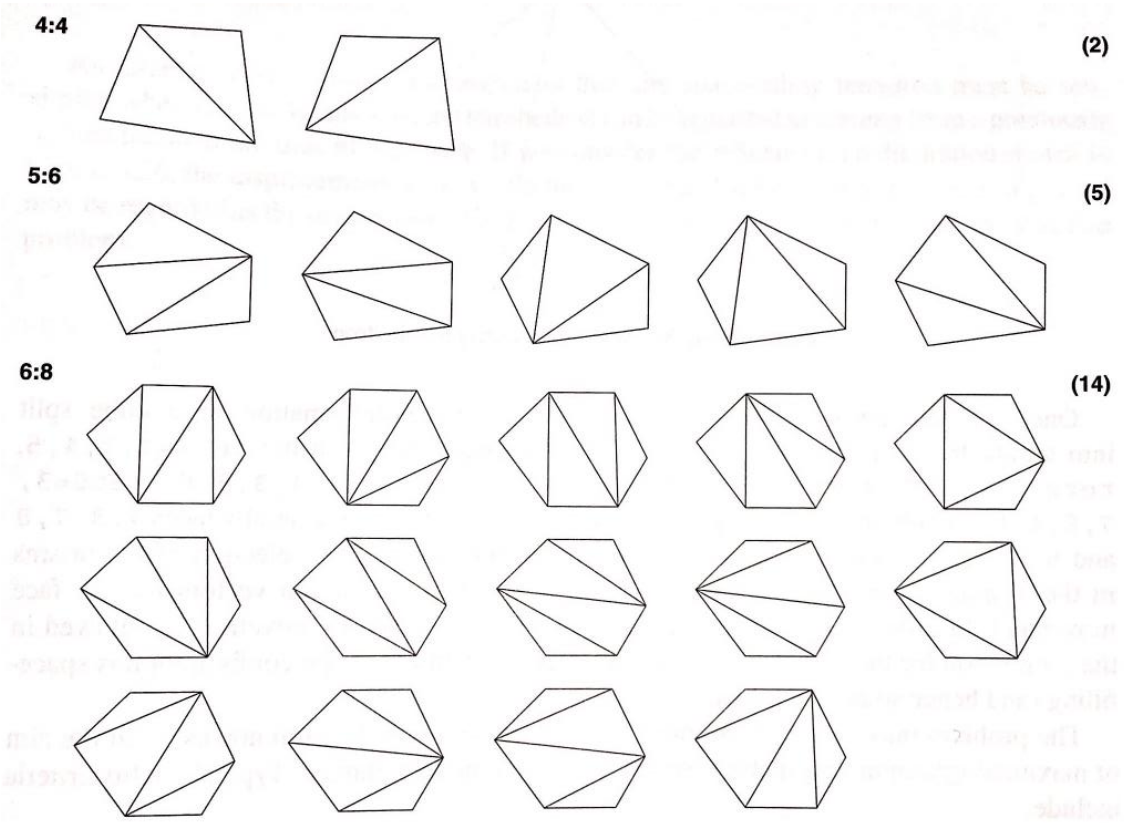


Figure 9 - Figure taken from Lhner(2008) which shows the many different options for Face-Swapping. The more cells that are involved gives more options for refinement.

To implement this the only logic that is required is the ability to determine whether the new cells have an average better cell quality than the previous. If they do have a higher quality, then keep the changes and move on to the next low quality cells.

2.5 Summary

It was found that the most robust method for implementing an unstructured grid generator in two dimensions was the Advancing Front Method. By taking the outline provided by Löhner and adding some of the additional cell formation techniques presented by Foucault et. al., a robust Advancing Front Algorithm can be developed.

Chapter 3

Methodology

This chapter will present the main advancing front algorithm and discuss each part of the code in detail so that it can be expanded upon and improved in future applications. Each function will be discussed in the order that they appear in the code because they are often called from within other functions at a later stage. The sections will each include a pseudocode of the function that will describe how they work in a general sense. It will also point to the specific lines of code where each function can be found in Appendix A.

To accurately demonstrate how each function works, there will be an example case of a 4x4 unit square grid-space as shown in Fig. 10. In situations where a written description is insufficient a visualization will be given based upon this example case.

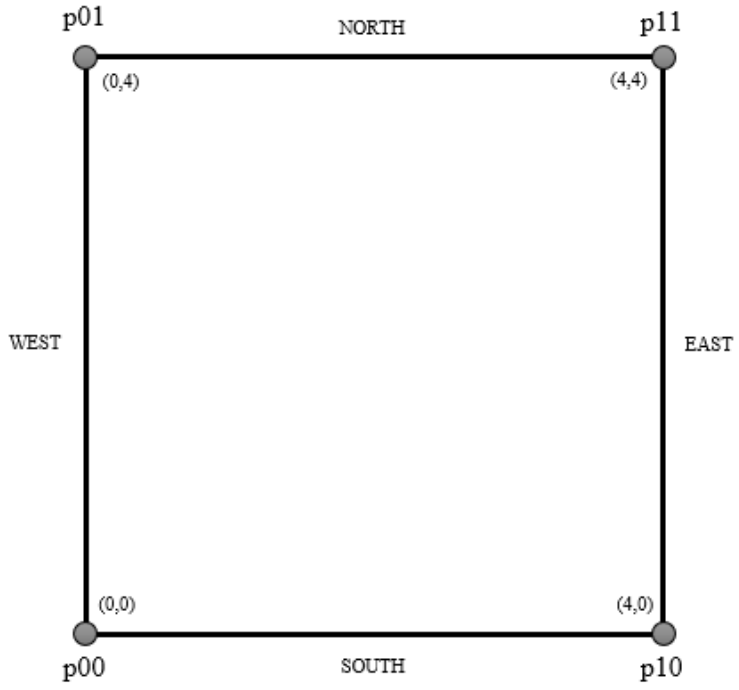


Figure 10 - Example 4x4 Grid-space.

3.1 Algorithm Overview

The advancing front algorithm that has been developed is based heavily upon the algorithm described by Löhner(2008)[1] with several additional mechanisms added to help reduce the chance for cell overlap. The most significant adjustments are the counter-clockwise marching of the advancing front, and the special cases that were implemented. In a general sense, the algorithm behaves as follows:

1. Define the boundaries of the grid-space. Keeping with Eilmer4 convention, the boundary is described from the bottom left point in a counter-clockwise fashion. In this case it will be describing the four points of a 2D quadrilateral test grid-space and interpolating the boundary points along these walls.
2. Create the list of these boundary points using their cartesian coordinates in a master list of points.
3. Connect these points to create the initial list of active faces. The order in which the points appear in the face's definition reflects the direction that the active face will propagate. The convention states that they should be described in a manner that has the propagation direction to the left of the direction of the face. This will be discussed in more detail in section 4.6.
4. Begin at the first entry of the active faces list and define what case it falls into; 0, 1, 2, 3.
5. Depending on the case, suggest a point to advance into the remaining grid-space. This can either be by defining a new point or using an existing point.
6. Create the new cell using this suggested point.
7. Clean up active faces, adding the new ones as required and removing ones that have been overwritten. Save the recently defined points, faces, and cells in the master lists.
8. If there are any remaining active faces loop back to part 4 for next indexed active face.

The greatest difference between this implementation and that of Löhner(2008)[1] is that instead of always advancing the shortest active face, it advances the next indexed face in the active face list. This was done to avoid a problem where the algorithm would focus on a boundary face that had a slightly shorter length because of a floating point rounding error, and would continuously build upon that face into the remaining grid-space. A representation of this can be found in Fig. 11.

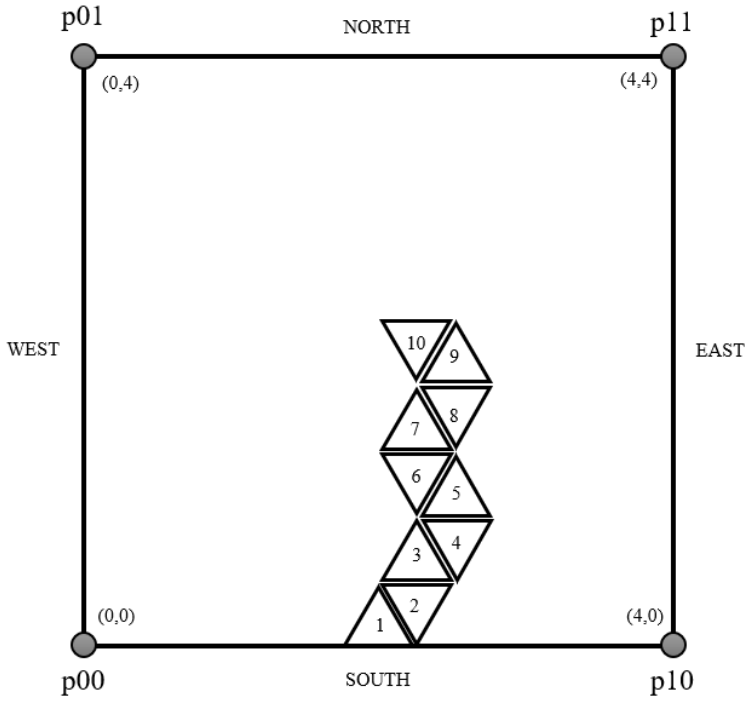


Figure 11 – The effect of a rounding error at the boundary propagating a chain of slightly smaller cells into the grid-space.

This created a significant amount of problems because the cells would not line up nicely when they reached the other boundaries. It created a clustering of small cells around the position where this long string of cells touched the other boundary and would continue to propagate from these smaller cells. For this reason, it was decided to enforce a counter-clockwise front propagation where possible.

Having this implementation however, would have an adverse effect on the gridding algorithm at later stages of the generation process. This was particularly nasty with concave grids where parts of the grid would be cut off from one another and no longer be connected by a single line of faces. To remedy this, four special cases were defined. The intention was to allow the gridding algorithm to cut corners and produce cell structures that actively avoid creating overlapping cells; in addition to avoiding the active faces from becoming segmented.

The four cases that were used are defined below, with a visual representation of each case shown in Fig. 12.

Case 0 is the regular face where there are no abnormal angles with the preceding or succeeding faces. In this instance it will attempt to create an orthogonal triangle cell using regular methods.

For Case 1, the current face encounters the next face in the list at an angle between 75° - 120°. The face then creates a split quadrilateral cell using both the current active face and the next active face.

Case 2 occurs when the current active face makes a suitable quadrilateral will the following two faces in the list. This case will overrule the requirements for case 1 so that it doesn't incorrectly duplicate the third face. The angle requirements for this case is making a total angle between 140° and 240° between the three active faces.

Case 3 is the final special case which occurs when there is a small angle between the current face and the next face in the list. The primary reason for this case is to avoid the current face from placing a new point over the top of the next face and creating an overlapping cell. To qualify for case 3 the angle between the two faces needs to be less than 60°.

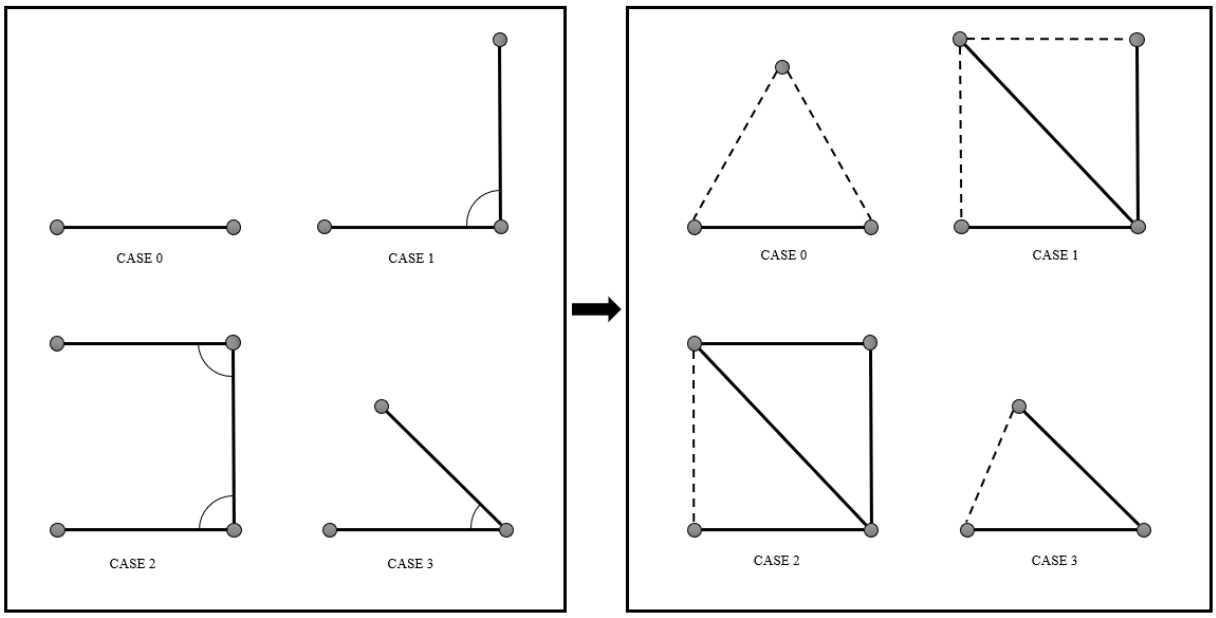


Figure 12 – Demonstration of how an active face propagates a cell based upon its special case.

The python code was then built around the framework of this algorithm. Each function will now be described in detail so that it can be reproduced.

3.2 *spacing* Function

The *spacing* function is used to determine a list of linear points along a boundary wall between two corner points. It is a placeholder for the input that it would receive from the Eilmer4 input script, that would dictate the geometry of the boundary and the number of cells it contains.

Given the limited complexity of the test cases, describing all walls as linear was considered sufficient.

The *spacing* function accepts three inputs; wall, scalar, and resolution. The wall input is the vector from the first corner point of the boundary to the second corner point. Scalar is the first point of the boundary which the function uses to scale the points. Finally, the resolution is simply the number of cells that is desired along the boundary wall.

The pseudocode for the function is as follows:

1. Determine the increment of the points (inverse of resolution)
2. Set up an empty list of boundary points
3. Iteratively step along the boundary by appending each new boundary point to the list using the function: $Point_i = ((wall_x \times increment_i + scalar_x), (wall_y \times increment_i + scalar_y))$
4. Return the list of boundary points

This resultant output is a list of boundary points excluding the final point. This is done so that there isn't a duplication of the corner points when walls are joined in the next function.

The code can be found in lines 11-24 of Appendix A.

3.3 *boundary_points* Function

The *boundary_points* function is used to create the full list of boundary points for the given grid-space. It accepts two inputs, a list of points and a list of resolutions. The function has been developed to allow for any number of points as long as they are described in a counter-clockwise fashion around the boundary. For instance, the example case in Fig. 10 would have to order the points as p00, p10, p11, p01. The list of resolutions is used to describe how many cells should appear on each of the boundary walls.

1. Create an empty list for the global boundary points
2. For each point in the input points list:
    I. Define the wall vector
    II. Define the resolution
    III. Define the starting point scalar
    IV. Obtain the boundary points from the *spacing* function
    V. Append the points to the global boundary points list

3. Reconstruct the list so that it's a continuous list of tuples
4. Return the full list of boundary points

The output from this is similar to that of the scalar function however it now contains the boundary points of the entire boundary rather than a single wall.

The code can be found lines 26-53 of Appendix A.

3.4 *initfaces* Function

The *initfaces* function takes in the boundary points that have been listed via the *boundary_points* function and turns them into faces. These faces are defined as tuples of points, based upon their index in the boundary list. For example, the 0^{th} face will be a tuple of (0, 1) referring to the 0^{th} and 1^{st} points in the boundary list, (0.0, 0.0), (0.4, 0.0). The pseudocode is:

1. Create an empty list of faces
2. Fear each point in the boundary point list, join it with the next point in the list to form a face
3. Append each new face to the list of faces
4. Return the list of initial faces

The code can be found in lines 55-75 of Appendix A.

3.5 *facelength* Function

The *facelength* function is used to obtain the length of a face by computing the magnitude of the vector between its points. It is used often when defining the suggested point for cell propagation, but also is useful when determining which faces have become too large.

1. Find the points of interest from the master points list
2. Determine the magnitude of the vector between the two points
3. Return the magnitude of the vector which is equal to the face length

The code can be found in lines 77-85 of Appendix A.

3.6 *suggestpoint* Function

The *suggestpoint* function is used to find an orthogonal point in the advancing direction of a given face. It does this by using some trigonometry to find the third point for the equilateral triangle given only the two points that make up the face. Take for example the face shown in Fig. 13 which has an active direction to the northwest.

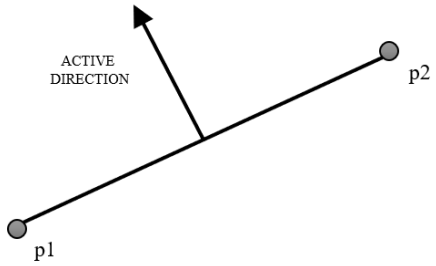


Figure 13 - Active face from p1 to p2 showing the direction in which the cell will propagate.

By finding the (x, y) vector of the change from point p1 to p2, the *suggestpoint* function can then use the following logic to find the ideal point which would make an orthogonal triangle.

$$\Delta x = (x_{p2} - x_{p1}), \qquad \Delta y = (y_{p2} - y_{p1})$$

$$halfpoint, h = \left(\frac{\Delta x}{2} \cdot x_{p1}, \frac{\Delta y}{2} \cdot y_{p1}\right)$$

$$p_3 = \left(\left(\frac{\sqrt{3}}{2} \cdot - \Delta y \cdot h_x\right), \left(\frac{\sqrt{3}}{2} \cdot \ \Delta x \cdot h_y\right)\right)$$

Figure 14 shows a graphical representation of what this looks like for the given example from Fig. 13.

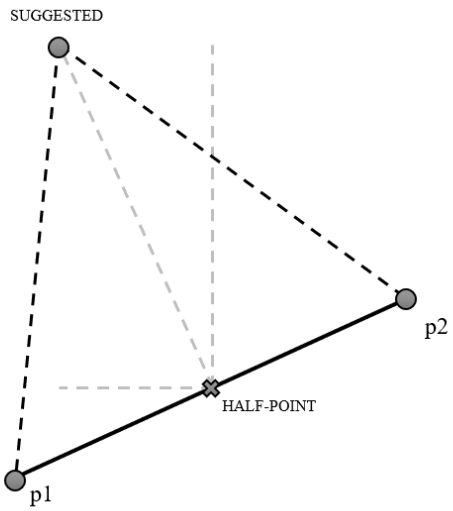


Figure 14 - Suggested Point based upon trigonometric algorithm.

This works for all orientations of faces and again reiterates the importance of describing the faces in the correct sequence. The pseudocode for this function is:

1. Obtain the (x,y) coordinates of the two points along the input face
2. Find the x-shift and the y-shift between the two points
3. Determine the half-point by adding half of the x and y shifts to the initial point
4. Find the suggested point by applying the function:

$$p_3 = \left(\left(\frac{\sqrt{3}}{2} \cdot -\Delta y \cdot h_x\right), \left(\frac{\sqrt{3}}{2} \cdot \Delta x \cdot h_y\right)\right)$$

5. Return the suggested point

The code can be found in lines 87-116 of Appendix A.

3.7 *newpoint* Function

The *newpoint* function handles the situations where the suggested point for the current active face can't be substituted by an existing point. This is only applicable to faces with cases 0 and 1. The following pseudocode is a coded representation of the actions described earlier in Fig. 12 from section 3.1.

1. If it's case 0:
    I. Append the suggested point
    II. Construct the two new faces made by the new point
    III. Append the new cell

IV. Remove the active face that created the cell
V. Append the new faces to the active faces list
VI. Append the new faces to the master faces list

2. If its case 1:
    I. Append the suggested point
    II. Construct the three new faces made by the new point
    III. Append the two cells that have been generated
    IV. Remove the current active face and the next active face that bred the new point
    V. Append the two new active faces to the active faces list
    VI. Append the three new faces to the master faces list

The codes can be found in lines 118-161 of Appendix A.

3.8 *existingpoint* Function

The *existingpoint* function serves a similar purpose to the *newpoint* function. Instead of forming a cell using a newly created point, it must form a cell with one of the existing points. For this reason, the function needs to be very robust in its approach to removing active faces and adding new ones; because there are many different situations where other active faces could make up the same new cell. If this isn't done correctly then there can be duplications of the faces which will inevitably crash the code. The pseudocode is shown below with the full code being available on lines 163-273 of Appendix A.

1. If its case 0:
    I. Create two new faces using the existing point
    II. Check the master faces list to see if those faces already exist; checking for both the new faces and the inverse of them
    III. Based upon whether these faces already existed, define the new cell bounded by the new faces where appropriate and existing faces where appropriate
    IV. Update the active faces list to remove the current active face and any active faces that already existed that have since been overwritten
2. If its case 1:
    I. Move the current active face to the end of the active face list
    II. Move the next active face to the end of the active face list; to preserve continuity and also to avoid getting stuck in a loop
3. If its case 2:

I. Create the two new faces as outlined in section 4.1
II. Define the new cells being generated
III. Append the new active face to active faces list
IV. Remove current active face; and next two active faces
V. Append new faces to master faces list

4. If its case 3:
   I. Create a new face sealing off the cell
   II. Check for it to be an existing face; checking for inverse as well
   III. Define the new cell and remove the next two active faces
   IV. Add the new face to the active faces list only if it doesn't already exist; add the face to the front of the list because it is likely to have a small face length

Most of the updates made to the active faces list and master lists of points, faces and cells are done through the *newpoint* and *existingpoint* functions. This frees up some room in the main function to allow for it to be more intuitive in its structure. It also ensures that the correct faces are being manipulated. This is because the cell is being constructed at the same time as the lists are being updated; so, there's no disconnect between the cell being made and the faces being deleted/formed.

3.9 *vect* Function

The *vect* function is very short and simple. Its purpose is to define an (x, y) vector between the two points in a face starting from the first point.

The code can be found in lines 275-284 of Appendix A.

3.10 *anglecheck* Function

The *anglecheck* function is used to determine the angles between the current active face, the previously indexed active face, and the next two indexed active faces. Its main objective is to help classify the case of the current active face into one of the four categories.

To do this the *anglecheck* function retrieves the vector of the current active face and the next two indexed active faces. It then uses this information in the atan2 in-built function that determines the angle between vectors and giving a result between $-\pi$ and π radians.

1. Obtain the vectors for the current face, next two faces, and previous face

2. Compute the angle between the previous face and the current face and save that as angle1
3. Compute the angle between the face i and face i+1 and save as angle2
4. Compute the angle between face i+1 and face i+2. Add this to angle2 to get the angles between all three faces. Save as angle3
5. Return the angle requested based upon the input call for the function.

The code can be found in lines 286-329 of Appendix A.

3.11 The Main Function

Having defined all the functions in the previous sections, the main function can now iteratively solve for an advancing front inside a boundary. The main function steps through the active face list while each time trying to define a new cell based upon what case it defines the active face to be. It tries to preserve the continuity of the active faces list so that it has no gaps that would affect the *anglecheck* function as it tries to find the angle between two faces that are not joined together. For this function the python code will be referenced directly.

Begin by defining the boundary of the grid-space using points in cartesian coordinates. Keeping with the example of the 4x4 square example case, the definition will be.

```
337 p0 = (0,0)
338 p1 = (4,0)
339 p2 = (4,4)
340 p3 = (0,4)
341
342 points = (p0, p1, p2, p3)
343 resolution = (10, 10, 10, 10)
344
345 boundaries = boundary_points(points, resolution)
```

So, the points are arranged in a counterclockwise fashion and the resolution is going to be the same on each of the walls. The boundary points are then defined using the *boundary_points* function of 4.3.

The next step is to define the initial active faces list and set up all the empty master lists of points, faces and cells.

```
363 master_points = boundaries
364 first_faces = initfaces(boundaries)
365
366 j=0
367 active_faces = first_faces
368 master_faces = initfaces(boundaries)
369 master_cells = []
```

Now the algorithm is ready to build the grid. The main building sequence takes place in a while loop which continues as long as there are active faces to build cells from. It begins by forcing the current active face to be the 0^{th} index of the active faces list. It then determines whether the face is too large to be advanced upon by making sure that its less than 4 times the length of the first active face. This is done to avoid having very large cells overlapping smaller cells and breaking the algorithm. If it is a large active face, then it is shoved to the end of the list and the next active face is chosen.

```
371 while len(active_faces) > 0:
372     i=0
373     case = 0
374     lengthoflist = len(active_faces)
375 
376     if facelength(active_faces[0], master_points) > 4.*facelength(master_faces[0], m
    aster_points):
377         shoveface = active_faces[0]
378         active_faces.remove(shoveface)
379         active_faces.append(shoveface)
```

In some rare fringe cases the algorithm has attempted to create faces between two of the same points resulting in an active face with a zero vector, which breaks the algorithm. After extensive debugging it was still not clear where this problem originated, so it was decided to simply remove these faces when they did arise in the main function.

```
380 
381     if active_faces[i+2][0] == active_faces[i+2][1]:
382         active_faces.remove(active_faces[i+2])
```

Now that the algorithm has an active face that it knows it is going to use to build the next cell, it must classify the face into its special case. This is done by using the *anglecheck* function described earlier to classify which of the four cases it will be. After some iterative debugging it was decided that it was also important to check the previous angles as well and classify them into their own special cases, 4 and 5.

```
383 
384     prev_ang = anglecheck(1)
385     first_ang = anglecheck(2)
386     second_ang = anglecheck(3)
387 
388     if 75. < first_ang < 120.:
389         case = 1
390     if 140. < second_ang < 240.:
391         case = 2
392     if first_ang < 60.:
393         case = 3
394     if prev_ang < 72.:
395         case = 4
396     if 75. < prev_ang < 112.:
397         case = 5
398     if case == 4:
399         facen1 = active_faces[lengthoflist-1]
400         active_faces.remove(facen1)
```

```
401         active_faces.insert(0, facen1)
402         case = 3
403     if case == 5:
404         facen1 = active_faces[lengthoflist-1]
405         active_faces.remove(facen1)
406         active_faces.insert(0, facen1)
407         if len(active_faces) <= 4:
408             case = 2
```

The angle definitions for cases 4 and 5 differ to those of their corresponding current cases 1,2 and 3, because they were found to work better with the algorithms build sequence. The angle cutoffs for each of the cases was chosen somewhat arbitrarily to achieve the best results for the largest amount of grid-space geometries. It should be noted that the cases are overwritten with each sequential *anglecheck* so that a case 1 which has ~90° angle with the next active face is overwritten by a case 2 which would also include an additional existing active face. This was done to prevent face duplication as much as possible. Case 4 essentially pulls the previous active face and performs a case 3 cell build on it to seal off the small angle cell. Case 5 pulls the previous active face and pushes the builder to its next iteration. The only time when it doesn't do this is when there are only 4 active faces left, because this means that these four faces are likely the remaining square that it has found; It then proceeds as a case 2.

Once the case has been determined the builder can begin deciding how to deal with the current active face. For case 0 it begins by finding the suggested point using the *suggestpoint* function and determining the distance it will search for an existing point. In the code this is called the variance.

```
410 if case == 0:
411         suggested = suggestpoint(active_faces[i], master_points)
412         suggested_fl = facelength(active_faces[i], master_points)
413         variance = suggested_fl/2.
414         """variance is just the distance in the x and y that it will
415         search for an existing point"""
416         result = [r for r in master_points if r[0]-
    variance <= suggested[0] <= r[0]+variance
417         and r[1]-variance <= suggested[1] <= r[1]+variance]
418
```

This works by searching for an existing point that is within the variance in both the x and y direction of the suggested point. Figure 15 shows an example of what this looks like. Due to a limitation in the list searching function, the first indexed point that meets the requirements of the search will be returned. For that reason, in the case shown in Fig. 15 the existing point 1 was chosen over existing point 2, even though the second point was the more favorable of the two.

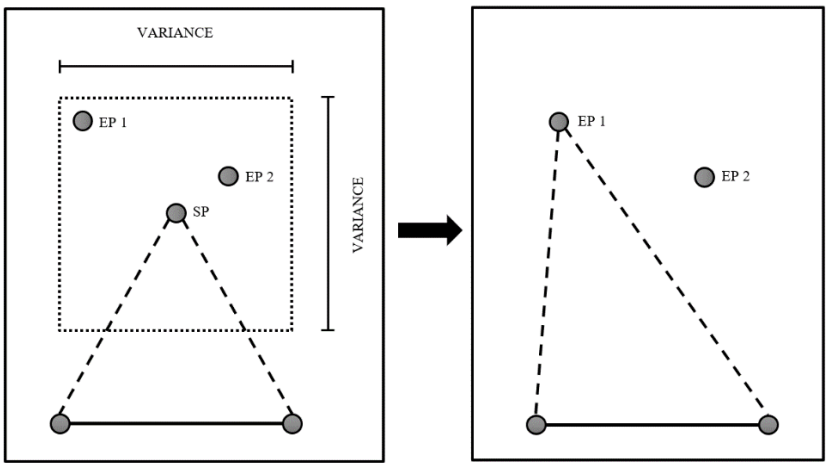


Figure 15 - Searching for an existing point using the variance. Indicating how the algorithm selects the first point in the existing points list even if it isn't the best match.

A similar process is used for case 1, however instead of suggesting a point that would make an orthogonal triangle it attempts to create an orthogonal square. It then divides this into two triangles because the algorithm can't handle cells with four faces.

```
419     elif case == 1:
420         suggested_fl = facelength(active_faces[i], master_points)
421         variance = suggested_fl/2.5
422         face_vector1 = vect(active_faces[0])
423         face_vector2 = vect(active_faces[1])
424 
425         face_vector1 = vect(active_faces[0])
426         face_vector2 = vect(active_faces[1])
427         face_vector1x = face_vector1[0]
428         face_vector1y = face_vector1[1]
429         face_vector2x = face_vector2[0]
430         face_vector2y = face_vector2[1]
431 
432         suggestedinit = master_points[active_faces[0][0]]
433         suggested = suggestedinit[0]+ face_vector2x, suggestedinit[1]+ face_vector2y
434 
435         result = [r for r in master_points if r[0]-
    variance <= suggested[0] <= r[0]+variance
436         and r[1]-variance <= suggested[1] <= r[1]+variance]
437 
```

For cases 2 and 3 the point that is going to be used to complete the cells is already known to be an existing point. In these cases, it forces the result to be this pre-determined existing point.

```
438     elif case == 2:
439         suggested = master_points[active_faces[2][1]]
440         result = [master_points[active_faces[i+2][1]]]
441 
442     elif case == 3:
443         suggested = master_points[active_faces[1][1]]
444         result = [master_points[active_faces[i+1][1]]]
445 
```

The following lines of code handle the case 5 loop skip and tell the builder to construct the cell for all other cases. It does this based upon whether a new point will be used, or if an existing point will suffice to use that instead.

```
446     if case == 5:
447         print "CASE 5"
448 
449     elif result == []:
450         newpoint(suggested, case)
451 
452     else:
453         result = result[0]
454         exist_point = master_points.index(result)
455         existingpoint(exist_point, case)
456 
```

It then loops back for the next active face. The break conditions for the algorithm are if there are fewer than two active faces, meaning that the final active face has nowhere to build to and the grid is completely built. It also has an iteration limit to avoid being stuck in a loop due to some error arising in the grid generation.

```
457     j+=1
458     print "Iteration:", j
459     if len(active_faces) <= 2:
460         print "Number of Cells:", len(master_cells)
461         break
462     if j > 1680:
463         print "Number of Cells:", len(master_cells)
464         print len(active_faces)
465         break
```

This completes the description of the advancing front algorithm. The full python code used to construct the 4x4 unit example case can be found in Appendix A. It can be run in any python IDE to display the generated mesh.

Chapter 4

Results

This chapter will present the results of the advancing front algorithm being applied to several different test cases. The aim is to test the algorithm's robustness to both changing geometries and grid resolutions. In most cases the test domain will be bounded between (0, 4) in both the x and y direction; with the boundary making up some polygon within these bounds.

The first two test cases were designed to investigate how the algorithm handled simple cases. It needed to be able to perform at this basic level before more complex geometries were introduced.

4.1 Initial 3x3 Square Mesh

The first test of the algorithm was a very simple 3x3 square boundary to test for any potential cell overlapping or cell duplication. It also tested the algorithm for any problems with resolving the grid when there were few faces remaining. The expectation was that the algorithm would divert to a continuous loop of cases 1 and 2 for the entirety of the solve. Figure 16 shows the generated grid.

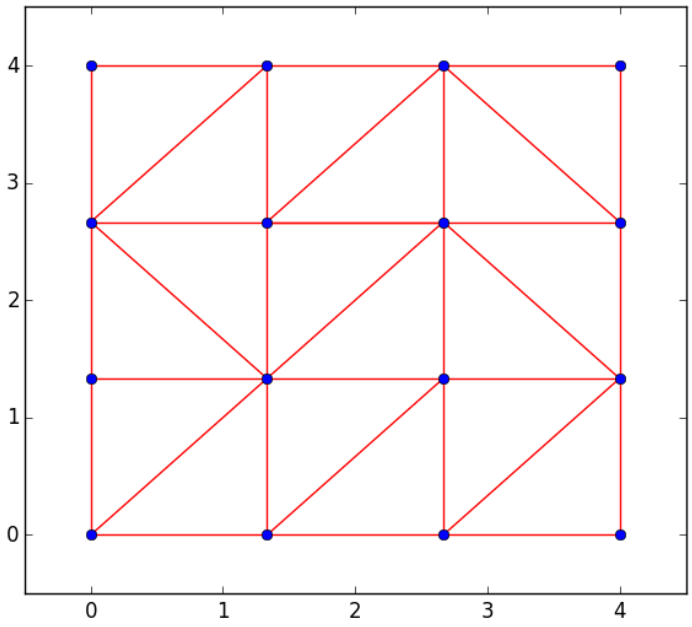


Figure 16 - 3x3 grid resolution square test case.

The result was a success, and the output contained a master list of cells of length 18, which is the exact number of the cells present in the grid. This means there has been no cell overlapping or cell duplicated; and the gridding algorithm is able to complete its loop correctly. Having completed this very simple case, the next test case increased the grid resolution to investigate how it handled the interior of the grid.

4.2 Higher Resolution 10x10 Square Mesh

The aim of this test case was to investigate how the algorithm handled the interior faces of the grid as it was being generated. Unlike the case in 4.1 where there was only boundary cells and a single interior cell, this case would have a significant number of interior cells. It mainly targeted the algorithms ability to correctly stitch together the front as it propagated further into the grid-space; and to check for any face duplications that may occur. The generated mesh can be found in Fig. 17.

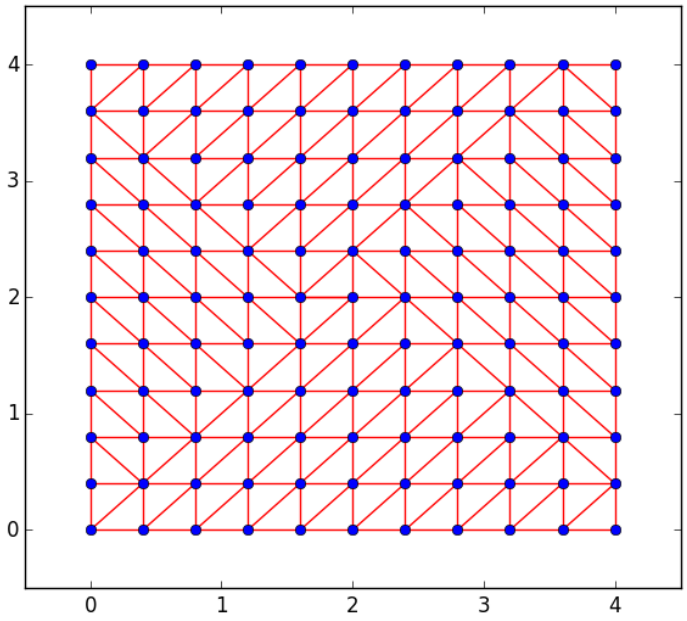


Figure 17 - 10x10 grid resolution square test case.

Similar to the 3x3 test case this grid completed using the correct terminations and produced the correct number of cells. The main concern is that it is only activating the 1st and 2nd special cases to grid this geometry. To fully justify the algorithm's competency, it needs to demonstrate that it can use all its special cases correctly without breaking.

4.3 Testing for Robustness to Geometry

It was important for the algorithm to be able to produce grids for complicated geometries as part of the fundamental reasons for unstructured grids. As outlined in chapter 2, unstructured grids are typically used in cases where structured grids would become too distorted as they are forced into concave geometries. To investigate the algorithms robustness to this, several different boundaries were designed. A set of four polygons were created to test how the algorithm handled relatively simple geometry changes.

4.3.1 Polygon Test Geometries

The polygons were variations upon the initial 4x4 unit test case used for 4.1 and 4.2, however in these cases they were distorted to have concave areas. Figure 18 shows the four polygons and their respective grids.

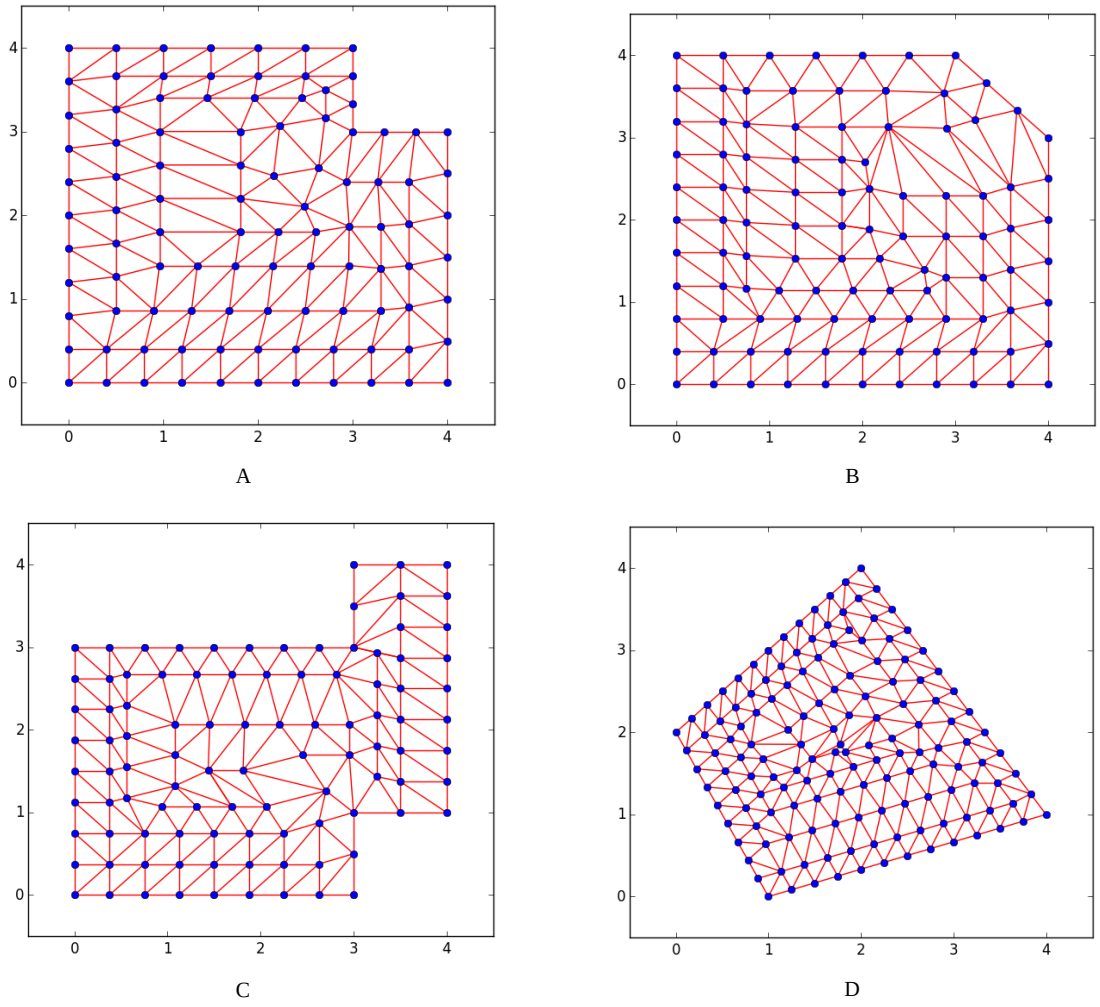


Figure 18 – The four polygon test cases that were used.

While the grids generated for these cases were successful, it was found that the algorithm was exceptionally sensitive to cell size changes. If the wall with the largest resolution contained an initial face that was approximately twice the size of the smallest resolution, the algorithm would eventually break. This was due to a cell propagating from the large faces overlapping many of the smaller cells created from the smallest resolution wall. This created active faces in the already gridded mesh which crashed the code.

It should also be noted that the cells in these grids were not of uniform size and would often border other cells that were significantly larger than they were. The transition of cell size should be gradual, otherwise the mesh will be prone to errors during the CFD simulation. The most glaring example of this was from Polygon D where there were three highly skewed cells propagating from a significantly smaller cell. Figure 19 shows this case and it can be seen that this would not form a suitable mesh for a CFD simulation.

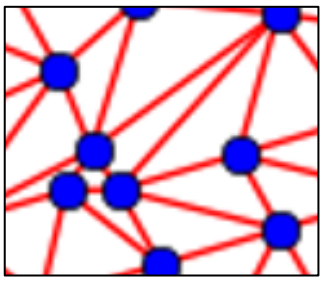


Figure 19 - Poorly generated cells from Polygon D.

4.3.2 Circular Test Geometry

Having moderate success gridding the polygon geometries it was decided that a circular boundary might give more promising results. The aim was to try and kick the algorithms reliance on forcing case 1 and 2 on the first group of boundary cells, which tended to propagate this pattern into the rest of the grid-space. It was anticipated that because the cells would each be propagating as a case 0, that the face lengths of each new active face would become smaller. It was expected that this would result in a grid with an exponentially smaller cell size as it propagated toward the center. This however, was not the case as the special cases 1 and 2 came into effect often enough to override the reducing cell size. Figure 20 shows the gridded circular boundary.

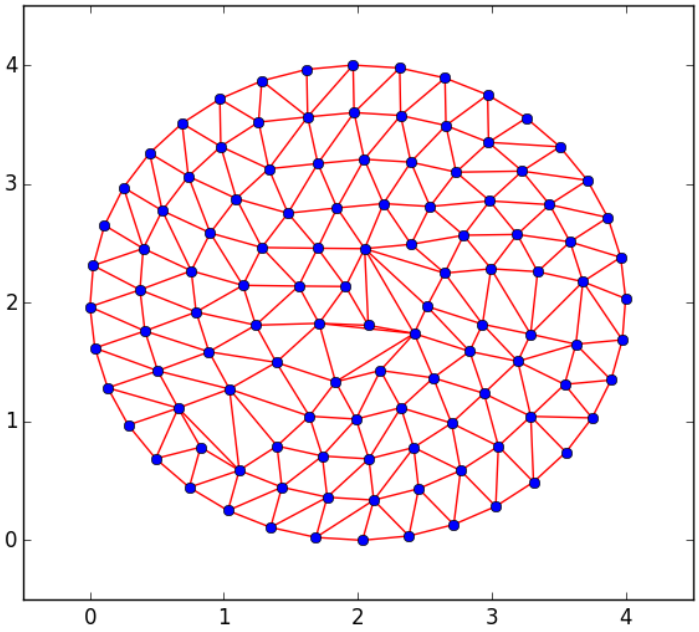


Figure 20 - Circular grid test case.

The grid generated was surprisingly uniform, however there was still a few issues with cell skewness and having small cells. Two main cases for the cell skewness are shown in Fig. 21 where there has been a large face snapping to an undesirable existing point, and then two cells have been formed to fill in the gaps around it.

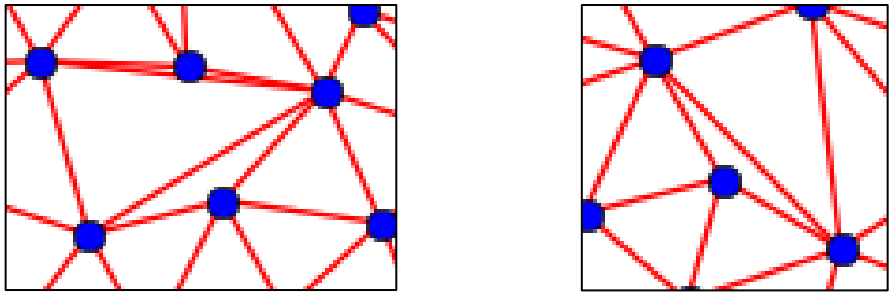


Figure 21 - Skewed cells found in Circular Grid.

There were a number of different causes for the cases of cell skewness and they will be discussed in Chapter 5. A full demonstration of the process undertaken to grid this circular mesh is present in Appendix B.

4.4 Hollow Grid Test Geometry

Hollow grids are common when running CFD simulations, especially when trying to define fluid flow over a body. Typically, the structured meshes will be described by both an internal boundary and an external boundary. The internal boundary is defined as a solid wall while the external boundary is described as either an inlet or an outlet for the flow, depending on how the simulation is set up. In the case for an unstructured grid it was imagined that this would remain the standard for describing the boundaries.

To facilitate this sort of boundary definitions a hollow grid geometry was designed that had both a bounded internal and external wall. The downside to this description however is that due to the way the algorithm is designed, it needs to have one continuous line of active faces to function. For this reason, the grid is essentially wrapped around the internal surface, with the seam at the center-south face. Figure 22 shows the initial hollow test case.

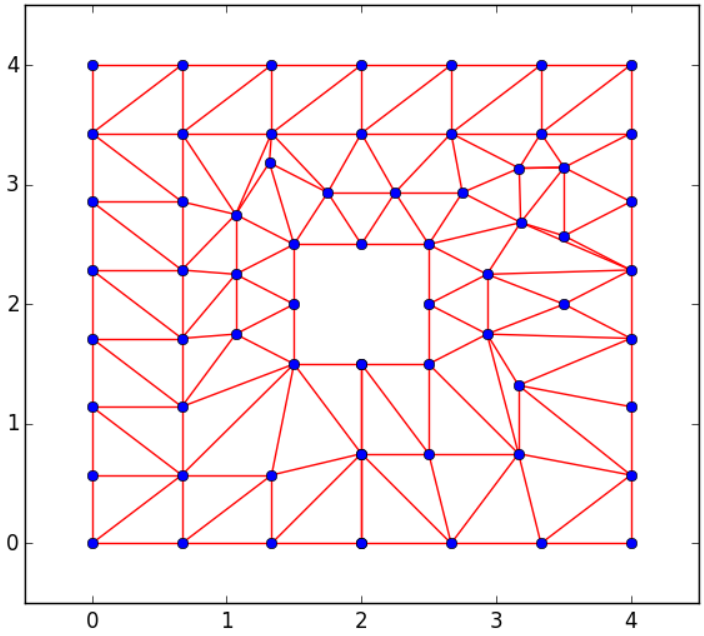


Figure 22 - Hollow grid test case with both an internal and external boundary.

This grid geometry was extremely sensitive to both grid size and the size of the internal hollow section. In this case the grid resolution was designed to be extremely coarse to minimize the amount of internal faces between the boundaries. This allowed the algorithm to solve for the grid geometry, but it is by no means able to run any meaningful simulation over this grid. For higher resolutions, the algorithm was simply incapable of keeping the active faces in a single long list. It would continuously segment the active faces into multiple pockets which would affect how the special cases worked, and would eventually lead to cell overlaps and code failure.

In another attempt, the thought process for how to describe a hollow grid was altered. For this case, it was assumed that only the inner boundary needed to be defined. The algorithm would be given essentially an inverted boundary and was told to continue to propagate this out until it broke. The result for this is shown in Fig. 23.

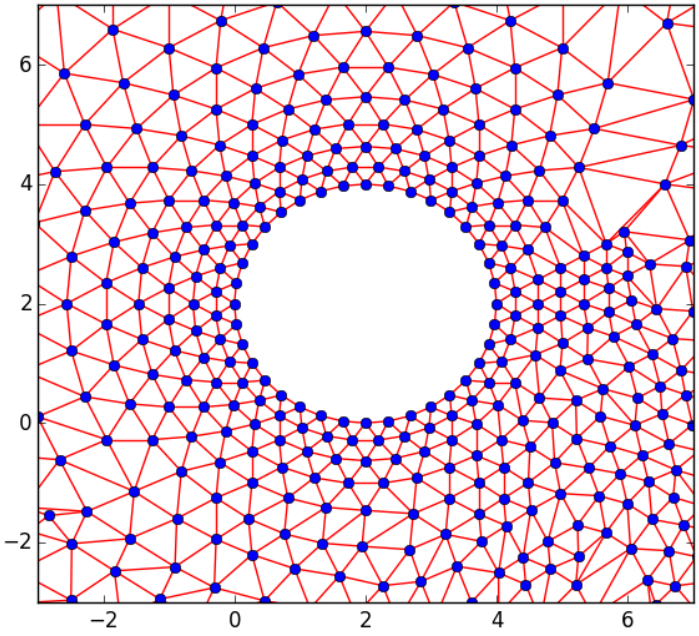


Figure 23 - Algorithm's attempt at creating an expansive flow field using only an internal boundary.

This type of hollow grid would be especially useful for producing outer flow-fields of aerofoil simulations. In this case it was a simple circle, however if it were able to line up with a structured chimera grid generated for an aerofoil's boundary layer, it could be useful. The downside for this gridding mechanism however is that the external boundary is very unpredictable, and wouldn't be easy to describe; which would limit its usability.

4.5 Bottle Grid Comparison

The final test case was one taken from literature. A test case was taken from the Eilmer4 Geometry User Guide[2] which demonstrated a structured grid around a bottle. In an effort to quantify the success of the algorithm, the grid geometry was replicated using the advancing front algorithm. It should be noted that due to the simplified boundary inputs that this algorithm takes, several of the Arc and Bezier boundaries used in the original Eilmer4 structured grid had to be replaced by straight line boundaries.

In addition, the graphical representation of the unstructured grid has been altered from the previous examples. The faces are shown as blue lines and the nodes are not displayed. This was to give a more intuitive way of visualizing the cell quality without the overrepresentation of nodes. Figures 24 and 25 show the mesh from literature and the generated unstructured mesh respectively.

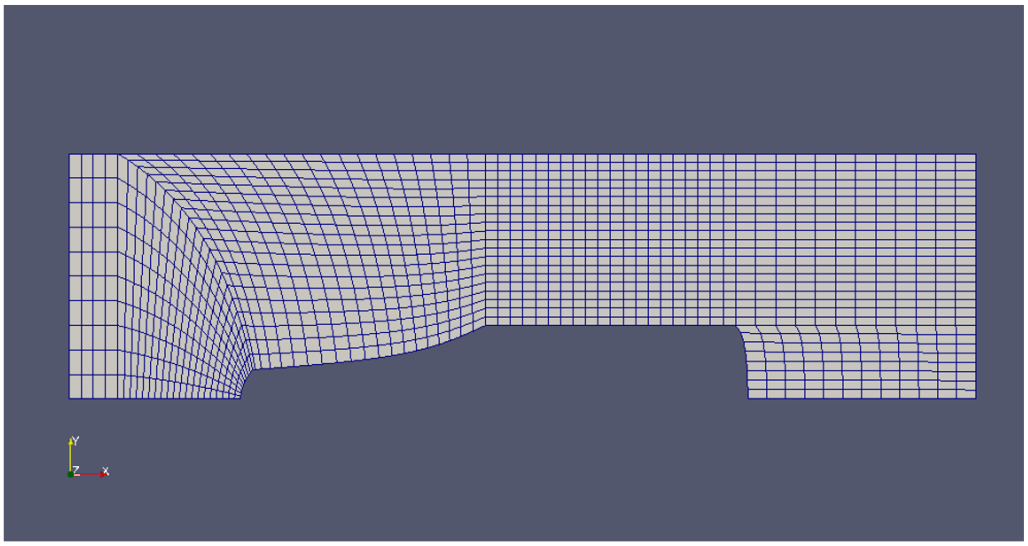


Figure 24 - Bottle Grid from Eilmer4 Geometry User Guide(2017).

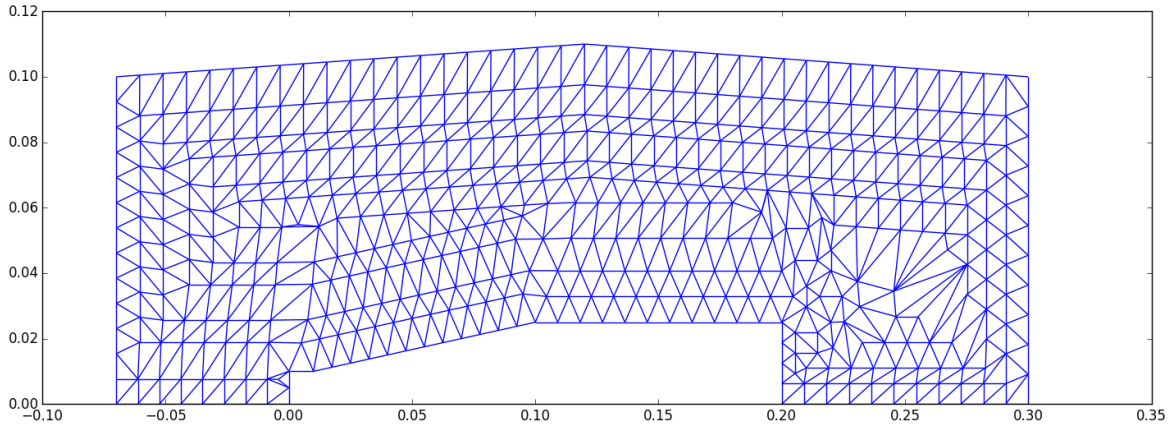


Figure 25 – Bottle Grid from Advancing Front Algorithm.

It can be seen from inspection that the unstructured mesh has several low-quality cells especially in the bottom right corner. In addition, the top boundary of the mesh had to be moved up slightly it order to give the algorithm enough room to grid; again, the problem was due to the concavity of the boundaries.

Chapter 5

Discussion and Recommendations

The implementation of the Advancing Front Algorithm has shown to be moderately successful. Multiple test cases could be gridded successfully, however the algorithm was very sensitive to the initial cell size. It was found that when one boundary met with another, and the initial face lengths along that boundary were roughly half that of the second boundary, the cells would overlap immediately during the first pass. This was due to the way the existing point search function was implemented, and will be covered in more detail in 5.1.

Implementing a version of the special cases approach found in Foucault et. al. proved to be a powerful technique for dealing with edge cases. Unfortunately, it began to be activated in situations where it was not intended. This is most clearly demonstrated in section 4.3.2 where the circular grid looks to have a seam running from the bottom left up to the center of the grid. This was propagated from an accidental activation of a case 1 face, which extended its face to be double the size of those around it. Once the face was this large, it would always form hard angles with the next faces in the front, and so the next face was forced to become a case 1 as well. It is recommended that this be changed in future improvements of this algorithm, because the large faces can potentially cause problems with cell overlap during the grids construction.

A result of particular interest was the algorithm's performance with the circular hollow grid. The grid quality of this mesh was far better than that of its double bounded counterpart from the previous section. This is a significant finding because of its application to gridding the outer flow-fields of aerofoils like that shown by Pointwise(2010). In this case the initial boundary of the unstructured grid was a circle. If it was instead seeded by a specifically designed structured grid for the boundary layer of aerofoil, then this could form a potentially powerful hybrid grid generator for objects in an expansive flow-field.

5.1 Reasons for Cell Skewness and Overlap

There were a handful of factors that went into creating highly skewed cells. The most prominent cause was through the poorly implemented search method for the existing points. When an existing point early in the list of points was found to satisfy the conditions in the search function, it would automatically be chosen. This happened regardless of whether it produced the most favorable cell structure. In future developments of this algorithm, the existing point should be searched for in an iterative fashion whereby the search domain is slowly increased. This would significantly reduce the risk for finding a poor existing point match.

Another contributor to low quality cells was special case 2. When determining whether a given face fell into this case, only the angles were checked to satisfy the conditions. In situations where the lengths of the joining faces were significantly different, then the cells that were being created were elongated and skewed. This was the case with the bottom right area of the bottle grid from section 4.5. By having a large variance in the length of the faces, the algorithm began to patch these together which created a series of long, skinny cells. A recommended fix for this would be the introduction of a face length checking tool that verified that the faces being connected in case 2 were of similar length.

5.2 Cell Refinement Strategies

The scope of this advancing front algorithm is only the initial part in a two-part technique for unstructured grid generation. The advancing front's purpose is to setup a rough outline for where the cells should be positioned; with a post-processing part to come in afterward and clean up the mesh. It is apparent that some of the cells that the algorithm produced were of significantly low quality. As outlined in Chapter 2 there does exist a number of potential refinement strategies. While it was outside the scope to implement these refinement methods, a manual attempt was made at the face swapping technique; to justify the importance of them. Take for example the distorted faces from the circular test geometry in 4.3.2. The first example showed an overextended cell that had produced two smaller cells in the gaps. By using Löhner's(2008) face swapping methods, the following refinement was obtained manually. Figure 26 shows this refinement.

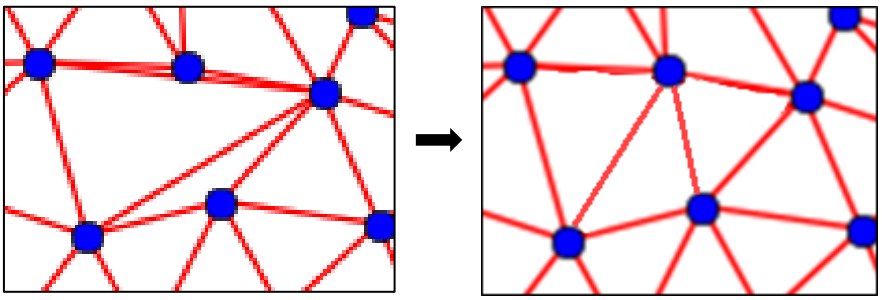


Figure 26 - Face-Swapping cell refinement using Löhner's methods outlined in section 2.

It should be noted that this is simply a representation of what a face-swapping algorithm would accomplish. The refinement was done manually through picture editing software.

Chapter 6

Conclusion

This thesis presents an extension on aspects of the advancing front method outlined in detail by Löhner(2008). It implemented several new mechanisms in an attempt to reduce the occurrence of cell overlap during grid generation and to produce a generally higher cell quality for the mesh. Although it didn't succeed at producing a completely robust grid generator, it has the potential to become one given further development. The algorithm has shown that it can resolve the active faces in a workable manner, and has a limited ability to mesh complex geometries.

The main limitations of the current approach concern its weakness and susceptibility to concave gridding geometries. By implementing the advancing front to propagate around the boundary walls, it essentially forced the algorithm to maintain a single list of faces which it would reference regularly. Relying on this list to determine which special case each face fell into, meant that when the front became segmented into pockets the algorithm would have a difficult time continuing. In addition to its difficulty with segmented fronts it also suffers from problems when finding suitable existing points. These problems are the most detrimental to the algorithms robustness and further work in these areas could overcome these weaknesses.

Future development of this algorithm and its research include:

- Developing a grid refinement strategy that would implement some of the grid refinement strategies outlined in section 2.4. This would allow the algorithm to produce high quality grids
- Implementing an iterative search function for existing points. It was acknowledged in section 5.1 that the current method for determining suitable existing points was flawed. If an iterative approach was taken whereby the search domain was increased from small to large, then the problem of always snapping to one of the earlier defined points would be minimized.

- Integrating a clustering function that would work alongside the boundary description to give a suggested face length for each cell. This would replace the current method of disallowing active faces with lengths four times greater than the initial cell from propagating.
- Extension to a mixed-cell algorithm. Having this algorithm's special cases would allow for the generation of quad cells during its build. It would be a relatively small step to implement quad cells in addition to the tri cells that this algorithm produces. This would assist with increasing the overall cell quality off the mesh and potentially increase efficiency when being solved in Eilmer4.

References

1 Löhner, R. (2008). *Applied computational fluid dynamics techniques*. Chichester, England: John Wiley & Sons, pp.35-107.

2 Dlang geometry package. (2017). Eilmer4 User Guide. Brisbane: School of Mechanical & Mining Engineering, pp.27-36.

3 Foucault, G., Cuillière, J., François, V., Léon, J. and Maranzana, R. (2007). An Extension of the Advancing Front Method to Composite Geometry. In: *16th International Meshing Roundtable*. Berlin: Springer, pp.288-312.

4 Chawner, J. (2017). *Quality and Control - Two Reasons Why Structured Grids Aren't Going Away*. [online] Pointwise. Available at: http://www.pointwise.com/theconnector/March-2013/Structured-Grids-in-Pointwise.shtml [Accessed 3 Sep. 2017].

5 Batts, J. (2010). *Latest Release of Pointwise CFD Mesher Features Hybrid Boundary Layer Meshing*. [online] Pointwise. Available at: http://www.pointwise.com/news/Pointwise-V1604-Released.shtml [Accessed 29 Aug. 2017].

6 Owen, S. (n.d.). Geometry Modeling & Grid Generation - ME469B/2/GI.

7 Mavriplis, D. (1997). UNSTRUCTURED GRID TECHNIQUES. *Annual Review of Fluid Mechanics*, 29(1), pp.473-514.

8 Bowyer, A. (1981). Computing Dirichlet tessellations. *Comput. J.*, 24(2), pp.162-166.

9 Wastson, DF. (1981). Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *Comput. J.*, 24(2), pp.167-171

10 Muir, H. (2017). *An Unstructured Mesh Generation Code for Eilmer4*. University of Queensland.

Appendices

Appendix A – Advancing Front Python Code

```
# -*- coding: utf-8 -*-
"""
Created on Wed Aug 16 13:17:15 2017

@author: M. Trudgian
"""
import math
import matplotlib.pyplot as plt
from matplotlib import collections as mc

"""
Defines the spacing of the boundaries by taking the
points and the resolution desired
"""
def spacing(wall,scalar,resolution):
    increment = 1./resolution
    i = 0
    boundi = []
    while i < resolution:               #its just < instead of <= to stop duplicatio
n of corner points
        a01 = i*increment
        pointi = ((wall[0]*a01 + scalar[0]),(wall[1]*a01 + scalar[1]))
        boundi.append(pointi)
        i+=1
    return boundi

"""
Outputs a list of (x,y) points that make up the
boundary of the block
"""
def boundary_points(points, resolution):
    bound_list = []
    i = 0
    lenp = len(points)
    while i < lenp:
        if i == lenp-1:
            wall = ((points[0][0]-points[i][0]), (points[0][1]-points[i][1]))
            res = resolution[i]
            startp = points[i]
            wallpoints = spacing(wall, startp, res)
            bound_list.append(wallpoints)
        else:
            wall = ((points[i+1][0]-points[i][0]), (points[i+1][1]-points[i][1]))
            res = resolution[i]
            startp = points[i]
            wallpoints = spacing(wall, startp, res)
            bound_list.append(wallpoints)
```

```
        i+=1
    i = 0
    bound_list2 = []
    while i < lenp:
        bound_list2 = bound_list2 + bound_list[i]
        i+= 1
    return bound_list2

"""
Takes the boundary points and creates a list of
((x,y),(x,y)) tuples which describes the faces

     ^
     |
p1 ---- p2

is the standard for the direction of the front
"""
def initfaces(boundary):
    initial_faces = []
    i=0
    while i < len(boundary):
        if i == len(boundary)-1:
            facei = (i, 0)
        else:
            facei = (i, i+1)
        initial_faces.append(facei)
        i+=1
    return initial_faces

"""
Just to find the length of the faces which
will come in handy when finding the face that
is smallest to use as the advancing face
"""
def facelength(p, pointlist):   #p is a face so two points
    p = (pointlist[p[0]], pointlist[p[1]])
    flength = math.sqrt((p[0][0]-p[1][0])**2 + (p[0][1]-p[1][1])**2)
    return flength

"""
This will suggest the point that will be an
equilateral triangle inward such that

         x
        / \
       /   \
      /     \
     /       \
   p1---------p2

where x is the suggested point

I think the point searcher, which will look for
another point that already exists that is in
the local area will be a sepearate function

It might even be better to just say that the
height should be equal to the halfway point so
that the corners are a bit nicer - will have to
look into this
"""
def suggestpoint(p, pointlist): #p is a face so two points
    p = (pointlist[p[0]], pointlist[p[1]])
    x = (p[1][0]-p[0][0])
    y = (p[1][1]-p[0][1])
    halfpoint = ((x/2.+p[0][0]),((y/2.+p[0][1])))
    scalar = math.sqrt(3)
```

```
    newpoint = (scalar*(-y/2)+halfpoint[0]),(scalar*(x/2)+halfpoint[1])
    return newpoint

"""
This newpoint adds the suggested points coordinates
to the master point list and uses its index to create
two new active faces.  It also removes the currently active
face from the active face list and updates the master
faces list
"""
def newpoint(suggested, case):
    if case == 0:
        master_points.append(suggested)
        last_point = len(master_points)-1
        newface1 = (active_faces[i][0], last_point)
        newface2 = (last_point, active_faces[i][1])

        master_cells.append([active_faces[i], newface1, newface2])

        active_faces.remove(active_faces[i])

        active_faces.append(newface1)
        master_faces.append(newface1)

        active_faces.append(newface2)
        master_faces.append(newface2)

    elif case == 1:
        master_points.append(suggested)
        last_point = len(master_points)-1
        newface1 = (active_faces[i][0], last_point)
        newface2 = (last_point, active_faces[i+1][1])
        newface3 = (active_faces[i][1], last_point)

        master_cells.append([active_faces[i], newface3, newface1])
        master_cells.append([active_faces[i+1], newface2, newface3])

        active_faces.remove(active_faces[i+1])
        active_faces.remove(active_faces[i])

        active_faces.append(newface1)
        master_faces.append(newface1)

        active_faces.append(newface2)
        master_faces.append(newface2)

        master_faces.append(newface3)

"""
existingpoint will set up the two possible faces that can exist
between the active face nodes and the existing point.  It checks
these and their inverses to see if they already exist within the
master_faces list so that they are not duplicated.  It will remove
active faces when it needs to for example when it's completing two
active faces at once.
"""
def existingpoint(exist_point, case):
    if case == 0:
        newface1 = (active_faces[i][0], exist_point)
        newface2 = (exist_point, active_faces[i][1])
        newface1inv = (exist_point, active_faces[i][0])
        newface2inv = (active_faces[i][1], exist_point)

        resultnew1 = [r for r in master_faces if r == newface1]
        resultnew1inv = [r for r in master_faces if r == newface1inv]
        resultnew2 = [r for r in master_faces if r == newface2]
        resultnew2inv = [r for r in master_faces if r == newface2inv]

```

```
        if resultnew1 != []:
            resultnew1 = resultnew1[0]
            exist_face1 = active_faces.index(resultnew1)
            active_faces.remove(active_faces[exist_face1])
            res1 = resultnew1

        elif resultnew1inv != []:
            resultnew1inv = resultnew1inv[0]
            exist_face1inv = active_faces.index(resultnew1inv)
            active_faces.remove(active_faces[exist_face1inv])
            res1 = resultnew1inv

        else:
            active_faces.append(newface1)
            master_faces.append(newface1)
            res1 = newface1

        if resultnew2 != []:
            resultnew2 = resultnew2[0]
            exist_face2 = active_faces.index(resultnew2)
            active_faces.remove(active_faces[exist_face2])
            res2 = resultnew2

        elif resultnew2inv != []:
            resultnew2inv = resultnew2inv[0]
            exist_face2inv = active_faces.index(resultnew2inv)
            active_faces.remove(active_faces[exist_face2inv])
            res2 = resultnew2inv

        else:
            active_faces.append(newface2)
            master_faces.append(newface2)
            res2 = newface2

        master_cells.append([active_faces[i], res1, res2])
        active_faces.remove(active_faces[i])

    elif case == 1:
        shoveface = active_faces[i]
        active_faces.remove(active_faces[i])
        active_faces.append(shoveface)

        shoveface = active_faces[i]
        active_faces.remove(active_faces[i])
        active_faces.append(shoveface)

        print "Error: Tried existingpoint for a newpoint (case1)"
        print "Face Shoved"

    elif case == 2:
        newface1 = (active_faces[i][0], exist_point)
        newface2 = (active_faces[i][1], exist_point)

        master_cells.append([active_faces[i], newface1, newface2])
        master_cells.append([active_faces[i+1], active_faces[i+2], newface2])

        active_faces.append(newface1)
        active_faces.remove(active_faces[i+1])
        active_faces.remove(active_faces[i+1])
        active_faces.remove(active_faces[i])

        master_faces.append(newface1)
        master_faces.append(newface2)

    elif case == 3:
        newface1 = (active_faces[i][0], exist_point)
        newface1inv = (exist_point, active_faces[i][0])

```

```
        resultnew1 = [r for r in master_faces if r == newface1]
        resultnew1inv = [r for r in active_faces if r == newface1inv]

        if resultnew1 != []:
            resultnew1 = resultnew1[0]
            exist_face1 = active_faces.index(resultnew1)
            master_cells.append([active_faces[i], active_faces[i+1], resultnew1])
            active_faces.remove(active_faces[exist_face1])
            active_faces.remove(active_faces[i+1])
            active_faces.remove(active_faces[i])
        elif resultnew1inv != []:
            resultnew1inv = resultnew1inv[0]
            exist_face1inv = active_faces.index(resultnew1inv)
            master_cells.append([active_faces[i], active_faces[i+1], resultnew1inv])

            active_faces.remove(active_faces[exist_face1inv])
            active_faces.remove(active_faces[i+1])
            active_faces.remove(active_faces[i])
        else:
            master_faces.append(newface1)
            master_cells.append([active_faces[i], active_faces[i+1], newface1])
            active_faces.remove(active_faces[i+1])
            active_faces.remove(active_faces[i])
            active_faces.insert(0, newface1)

"""
Gives a vector (not normalised) based on the active faces points
"""
def vect(facepoints):
    i01x = master_points[facepoints[0]][0]
    i01y = master_points[facepoints[0]][1]
    i02x = master_points[facepoints[1]][0]
    i02y = master_points[facepoints[1]][1]
    vec0 = i02x-i01x, i02y-i01y
    return vec0

"""
Basically just a dot product to find the angles between faces
by turning them into vectors and using that
"""
def anglecheck(num_faces):
    i0 = active_faces[0]
    i1 = active_faces[1]
    i2 = active_faces[2]
    lastpoint = len(active_faces)-1
    in1 = active_faces[lastpoint]

    vec0 = vect(i0)
    vec1 = vect(i1)
    vec2 = vect(i2)
    vecn1 = vect(in1)

    angle1 = 180. - round((math.atan2(vec0[1], vec0[0])-
math.atan2(vecn1[1], vecn1[0]))*(180./math.pi),4)
    if angle1 > 360:
        angle1 = angle1 - 360
    if angle1 < 0:
        angle1 = angle1 + 360

    angle2 = 180. - round((math.atan2(vec1[1], vec1[0])-
math.atan2(vec0[1], vec0[0]))*(180./math.pi),4)
    if angle2 > 360:
        angle2 = angle2 - 360
    if angle2 < 0:
        angle2 = angle2 + 360

    angle3 = 180. - round((math.atan2(vec2[1], vec2[0])-
math.atan2(vec1[1], vec1[0]))*(180./math.pi),4)
```

```
    if angle3 > 360:
        angle3 = angle3 - 360
    if angle3 < 0:
        angle3 = angle3 + 360

    angle3 = angle3 + angle2

    if num_faces == 1:
        return angle1

    elif num_faces == 2:
        return angle2

    elif num_faces == 3:
        return angle3

'''''
Describe your boundaries here using a points list in a counterclocwise fashion
Don't forget to alter your resolutions, each entry corresponds to that face
and try to make them be of roughly the same size.  Perhaps create a function
that will auto-fill the resolution based on the grid sizing that you'd like
'''
p0 = (0,0)
p1 = (4,0)
p2 = (4,4)
p3 = (0,4)

points = (p0, p1, p2, p3)
resolution = (10, 10, 10, 10)

boundaries = boundary_points(points, resolution)

'''''
Used for describing the circle instead of using the boundary_points function
'''

'''''
boundaries = []
tt = 1.
while tt < 360:
    ttrad = (tt*math.pi)/(180.)
    xp = 2*math.cos(ttrad) + 2
    yp = 2*math.sin(ttrad) + 2
    pointint = (xp, yp)
    boundaries.append(pointint)
    tt+=10
'''

master_points = boundaries
first_faces = initfaces(boundaries)

j=0
active_faces = first_faces
master_faces = initfaces(boundaries)
master_cells = []

while len(active_faces) > 0:
    i=0
    case = 0
    lengthoflist = len(active_faces)

    if facelength(active_faces[0], master_points) > 4.*facelength(master_faces[0], m
aster_points):
        shoveface = active_faces[0]
        active_faces.remove(shoveface)
        active_faces.append(shoveface)

    if active_faces[i+2][0] == active_faces[i+2][1]:
```

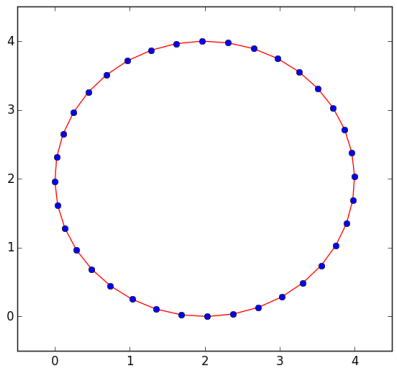
```
382         active_faces.remove(active_faces[i+2])
383 
384     prev_ang = anglecheck(1)
385     first_ang = anglecheck(2)
386     second_ang = anglecheck(3)
387 
388     if 75. < first_ang < 120.:
389         case = 1
390     if 140. < second_ang < 240.:
391         case = 2
392     if first_ang < 60.:
393         case = 3
394     if prev_ang < 72.:
395         case = 4
396     if 75. < prev_ang < 112.:
397         case = 5
398     if case == 4:
399         facen1 = active_faces[lengthoflist-1]
400         active_faces.remove(facen1)
401         active_faces.insert(0, facen1)
402         case = 3
403     if case == 5:
404         facen1 = active_faces[lengthoflist-1]
405         active_faces.remove(facen1)
406         active_faces.insert(0, facen1)
407         if len(active_faces) <= 4:
408             case = 2
409 
410     if case == 0:
411         suggested = suggestpoint(active_faces[i], master_points)
412         suggested_fl = facelength(active_faces[i], master_points)
413         variance = suggested_fl/2.
414         """variance is just the distance in the x and y that it will
415         search for an existing point"""
416         result = [r for r in master_points if r[0]-
    variance <= suggested[0] <= r[0]+variance
417         and r[1]-variance <= suggested[1] <= r[1]+variance]
418 
419     elif case == 1:
420         suggested_fl = facelength(active_faces[i], master_points)
421         variance = suggested_fl/2.5
422         face_vector1 = vect(active_faces[0])
423         face_vector2 = vect(active_faces[1])
424 
425         face_vector1 = vect(active_faces[0])
426         face_vector2 = vect(active_faces[1])
427         face_vector1x = face_vector1[0]
428         face_vector1y = face_vector1[1]
429         face_vector2x = face_vector2[0]
430         face_vector2y = face_vector2[1]
431 
432         suggestedinit = master_points[active_faces[0][0]]
433         suggested = suggestedinit[0]+ face_vector2x, suggestedinit[1]+ face_vector2y

434 
435         result = [r for r in master_points if r[0]-
    variance <= suggested[0] <= r[0]+variance
436         and r[1]-variance <= suggested[1] <= r[1]+variance]
437 
438     elif case == 2:
439         suggested = master_points[active_faces[2][1]]
440         result = [master_points[active_faces[i+2][1]]]
441 
442     elif case == 3:
443         suggested = master_points[active_faces[1][1]]
444         result = [master_points[active_faces[i+1][1]]]
445 
446     if case == 5:
```

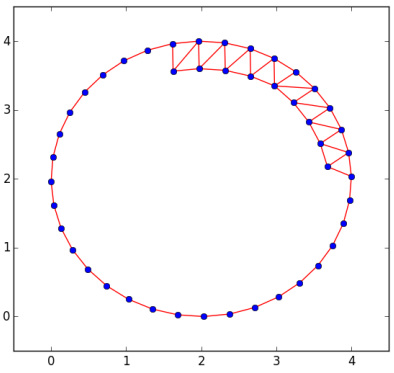
```
447         print "CASE 5"
448 
449     elif result == []:
450         newpoint(suggested, case)
451 
452     else:
453         result = result[0]
454         exist_point = master_points.index(result)
455         existingpoint(exist_point, case)
456 
457     j+=1
458     print "Iteration:", j
459     if len(active_faces) <= 2:
460         print "Number of Cells:", len(master_cells)
461         break
462     if j > 1680:
463         print "Number of Cells:", len(master_cells)
464         print len(active_faces)
465         break
466 
467 zz2 = zip(*master_points)
468 
469 lines = []
470 for i in master_faces:
471     lines.append([(master_points[i[0]][0], master_points[i[0]][1]), (master_points[i
    [1]][0], master_points[i[1]][1])])
472 
473 lc = mc.LineCollection(lines, colors=[(1, 0, 0, 1)], linewidths=1)
474 fig, ax = plt.subplots()
475 ax.add_collection(lc)
476 
477 ax.set_xlim([-0.5, 4.5])
478 ax.set_ylim([-0.5, 4.5])
479 plt.plot(zz2[0], zz2[1], 'bo')
480 plt.show()
481 
482 """
483 This isn't as robust as it probably should be.  I cannot overstate how extremely
484 sensitive it is to the initial grid size, because it is very very likely that at
485 some point in the late stages of the gridding process that you will overlap your
486 cells and produce problems.  I tried as best I could to avoid this but as you will
487 likely notice for complex geometries and varying grid sizes, the larger grids will
488 almost always overstep into the inactive area of the grid, causing it to crash.
489 
490 What might be worth looking into is an iterative search for the closest point
491 in a process similar to this:
492 
493 ----------------------
494 |   --------------   |
495 |   |   ------   |   |
496 | 3 | 2 | 1. |   |   |
497 |   |   ------   |   |
498 |   --------------   |
499 ----------------------
500 
501 With each zone an iterative search domain.  Because at the moment it just searches
502 the whole domain and finds the first point in the master_points list that satisfies
503 the search variance.  This is usually a point thats closer to the boundary just by
504 definiition of how the AF gridder operates.
505 
506 Another thing that might be useful is trying to instead search for the front that
507 has the smallest facelength and shoving everything that comes before it to the back
508 of the list.  This is done in small quantities in some parts of this algorithm when
509 large faces came up that would produce problems.  By shoving them to the end of the
510 active_faces list it preserves the order in which the faces are connected but also
511 allows you to minimise cell overlap.
```

```
512
513 It is also important to recognise how poorly this gridder handles two advancing fron
    t
514 domains.  So say if you section each domain off by a gridded space, the list of face
    s
515 will have a point where it claims one face is connected to a face in a completely
516 different part of the gridspace.  This is particularly problematic when the angles
517 between the two faces qualifies it to be a special case and the AF gridder tries
518 to connect them with a square cell or something similar.
519
520 In any case this should be a good starting point for an AF gridder for Eilmer4
521 and some additional post processing could end up making it quite robust.
522 """
```

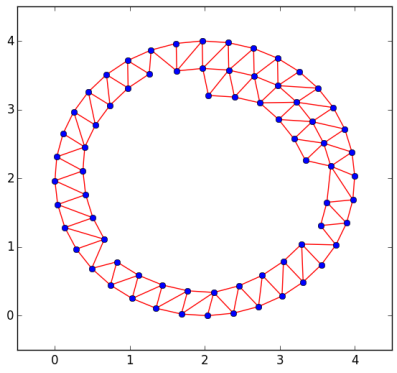
Appendix 2 – Circular Grid Generation



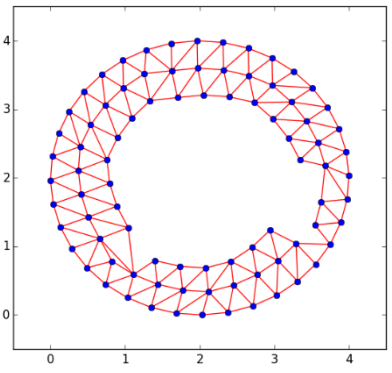
1



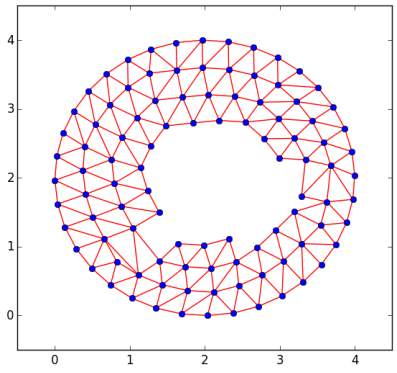
2



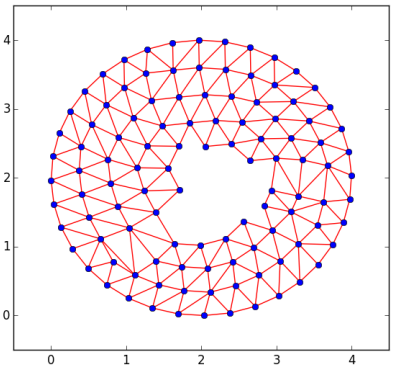
3



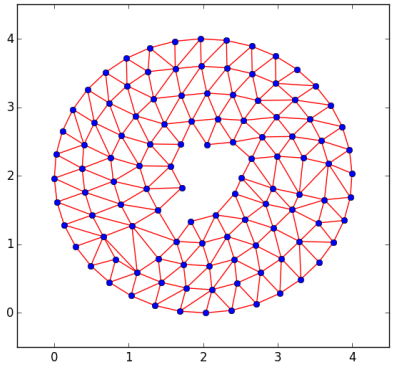
4



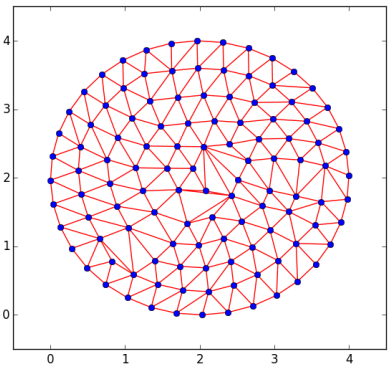
5



6



7



8